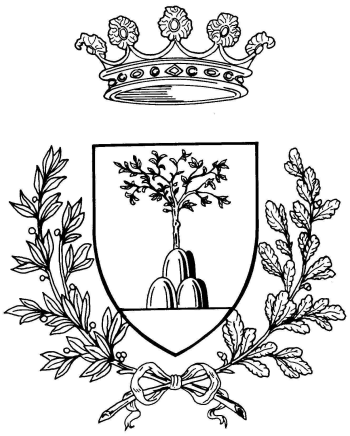
Università degli Studi di Ferrara

Dipartimento di Ingegneria



DOTTORATO DI RICERCA IN SCIENZE DELL'INGEGNERIA

Ciclo XXX

COORDINATORE Prof. Stefano Trillo

SETTORE SCIENTIFICO DISCIPLINARE ING-INF/05

Inference and Learning Systems for Uncertain Relational Data

Dottorando
Dott. Giuseppe Cota

Tutori
Prof.ssa Evelina Lamma
Prof. Fabrizio Riguzzi

Anni: 2014/2017

Abstract

Representing uncertain information and being able to reason on it is of foremost importance for real world applications. The research field *Statistical Relational Learning* (SRL) tackles these challenges. SRL combines principles and ideas from three important subfields of Artificial Intelligence: machine learning, knowledge representation and reasoning on uncertainty. The *distribution semantics* provides a powerful mechanism for combining logic and probability theory.

The distribution semantics has been applied so far to extend Logic Programming (LP) languages such as Prolog and represents one of the most successful approaches of Probabilistic Logic Programming (PLP), with several PLP languages adopting it such as PRISM, ProbLog and LPADs. However, with the birth of the Semantic Web, that uses Description Logics (DLs) to represent knowledge, it has become increasingly important to have *Probabilistic Description Logic* (PDLs). The DISPONTE semantics was developed for this purpose and applies the distribution semantics to description logics.

The main objective of this dissertation is to propose approaches for reasoning and learning on uncertain relational data. The first part concerns reasoning over uncertain data. In particular, with regard to reasoning in PLP, we present the latest advances in the `cplint` system, which allows hybrid programs, i.e. programs where some of the random variables are continuous, and causal inference. Moreover `cplint` has a web interface, named `cplint` on SWISH, which allows the user to easily experiment with the system. To perform inference on PDLs that follow DISPONTE, a suite of algorithms was developed: BUNDLE ("Binary decision diagrams for Uncertain reasoNing on Description Logic thEories"), TRILL ("Tableau Reasoner for descrIption Logics in Prolog" and TRILL^P ("TRILL powered by Pinpointing formulas").

The second part, which focuses on learning, considers two problems: *parameter learning* and *structure learning*. We describe the systems EDGE ("Em over bDds for description loGics paramEter learning") for parameter learning and LEAP ("LEArning Probabilistic description logics") for structure learning of PDLs. The execution of these algorithms and those for PLP, such as EMBLEM for parameter learning and SLIPCOVER for structure learning, is rather expensive from a computational point of view, taking a few hours on datasets of the order of MBs. In order to efficiently manage larger datasets in the era of Big Data and Linked Open Data, it is extremely important to develop fast learning algorithms. One solution is to distribute the algorithms using modern computing infrastructures such as clusters and clouds. We thus extended EMBLEM, SLIPCOVER, EDGE and LEAP to exploit these facilities by developing their MapReduce versions: EMBLEM$^{\text{MR}}$, SEMPRE, EDGE$^{\text{MR}}$ and LEAP$^{\text{MR}}$.

We tested the proposed approaches on real-world datasets and their performance was comparable or superior to those of state-of-the-art systems.

Acknowledgements

First and foremost, I have to thank my two supervisors, Evelina Lamma and Fabrizio Riguzzi. I am immensely grateful for all the time they dedicated to me. Their guidance and support were essential during my PhD years.

I would like to thank my colleagues. Thanks to the interesting discussions and the fun we had together, the cafeteria food and the sandwiches seemed to be tastier.

I thank all the components of my family that helped me to grow up and become an adult. In particular, I thank my father Matteo, my mother Maria and my brother Antonio for forcing me to believe in myself even when I did not want to. I also thank my grandparents Antonio and Anna for their encouragement throughout my doctoral years.

A special thank goes to Paola, the talking chicken who interrupts my grouchiness (often without permission) and fills my days with colourful stories and crayons.

Last but not least, I need to thank all my friends for all the marvelous moments of joy spent together. Without them I would have published more papers.

Giuseppe Cota

And now for something completely different...

Contents

List of Figures **XV**

List of Tables **XVII**

List of Algorithms **XIX**

List of Acronyms **XXI**

I Introduction **1**

1 Motivation **3**

2 Aims of the Thesis **7**

3 Structure of the Thesis **9**
3.1 Structure 9
3.2 Thesis Contributions 10
3.2.1 Inference in Probabilistic Logic Programming 10
3.2.2 Inference in Probabilistic Description Logics 11
3.2.3 Learning Systems in Probabilistic Logic Programming 11
3.2.4 Learning Systems in Probabilistic Description Logics 12
3.3 How to read this thesis 12

II Probabilistic Logics **15**

4 Fundamentals of First-Order Logic and Logic Programming **17**
4.1 Introduction 17
4.2 First-Order Logic 17
4.2.1 Syntax 17
4.2.2 Tarski's semantics 20
4.3 Logic Programming 21
4.3.1 Prolog 22
4.3.2 Normal Logic Programs 24
4.4 First-Order Logic vs Logic Programs 28
4.5 Conclusions 29

5 Distribution Semantics **31**
5.1 Introduction . . . 31
5.2 Formal Definition . . . 32
5.3 Conclusions . . . 34

6 Probabilistic Logic Programming Languages **35**
6.1 Introduction . . . 35
6.2 Logic Programs with Annotated Disjunctions . . . 35
6.2.1 LPADs Syntax . . . 35
6.2.2 LPADs Semantics . . . 36
6.3 ProbLog . . . 39
6.3.1 ProbLog Syntax . . . 39
6.4 Conclusions . . . 40

7 Description Logics and OWL **41**
7.1 Introduction . . . 41
7.2 Description Logics . . . 42
7.3 Syntax . . . 42
7.3.1 Concept and Role Constructors . . . 43
7.3.2 Concept Constructors . . . 43
7.3.3 Role constructors . . . 44
7.3.4 Knowledge Base . . . 45
7.3.5 Nomenclature . . . 47
7.4 Semantics . . . 50
7.4.1 Decidability of Description Logics . . . 54
7.5 Description Logics and First-Order Logic . . . 55
7.6 The OWL Ontology Language . . . 57
7.6.1 OWL Syntax . . . 59
7.6.2 OWL sublanguages . . . 60
7.6.3 Tools for OWL . . . 63
7.7 Conclusions . . . 63

8 Reasoning in Description Logics **65**
8.1 Reasoning Problems . . . 65
8.1.1 Closed vs Open World Assumption . . . 67
8.2 Reasoning Techniques . . . 67
8.2.1 Pellet . . . 68
8.2.2 Tableau Algorithm . . . 68
8.2.3 Explanation finding . . . 72
8.2.4 Pinpointing formula . . . 80
8.3 Conclusions . . . 84

9 Probabilistic Description Logics **85**
9.1 Introduction . . . 85
9.2 The Distribution Semantics for Description Logics: DISPONTE . . . 85
9.2.1 Syntax . . . 85
9.2.2 Semantics . . . 86

9.2.3 Assumption of Independence 90
9.3 Related Work 92
9.4 Conclusions 94

III Inference in Probabilistic Logics 95

10 Decision Diagrams 97
10.1 Introduction 97
10.2 Multivalued Decision Diagrams 97
10.3 Binary Decision Diagrams 98
10.4 Conclusions 100

11 Fundamentals of Exact Probabilistic Logical Inference 101
11.1 Inference Approaches 101
11.2 Exact Probabilistic Logical Inference 102
11.3 Splitting Algorithm 104
11.4 Inference with Multi-valued Decision Diagrams 107
11.5 Inference with Binary Decision Diagrams 108
11.6 Conclusions 114

12 Inference in Probabilistic Logic Programming 115
12.1 Introduction 115
12.2 `cplint` 116
12.2.1 Exact Inference: the PITA module 116
12.2.2 Approximate Inference: the MCINTYRE module 118
12.3 Causal Inference with `cplint` 121
12.3.1 Causal Inference in Probabilistic Logic Programming 124
12.3.2 Causal Exact Inference with `cplint` 125
12.3.3 Causal Approximate Inference with `cplint` 126
12.3.4 Notable Examples 127
12.3.5 Simpson's Paradox 127
12.3.6 Viral Marketing 129
12.3.7 Experiments 129
12.4 Hybrid Probabilistic Logic Programs with `cplint` 136
12.4.1 Sampling the Arguments of Unconditional Queries over Hybrid Programs 137
12.4.2 Conditional Queries over Hybrid Logic Programs 138
12.5 `cplint` on SWISH: a Web interface for `cplint` 141
12.5.1 SWISH 141
12.5.2 `cplint` on SWISH 143
12.5.3 Examples 146
12.6 Related Work 152
12.6.1 Work on causality inference 152
12.6.2 Work on Hybrid Probabilistic Logic Programs 153
12.6.3 Web application for Probabilistic Logic Programming 153
12.7 Conclusions 154

13 Inference in Probabilistic Description Logics 155
13.1 Introduction . . . 155
13.2 BUNDLE . . . 156
13.2.1 How to use BUNDLE . . . 158
13.3 TRILL . . . 158
13.4 TRILL^P . . . 165
13.5 How to use TRILL and TRILL^P . . . 166
13.6 TRILL on SWISH . . . 166
13.7 Inference Complexity . . . 167
13.8 Experiments . . . 168
13.8.1 Comparing the Systems . . . 168
13.9 Related Work . . . 172
13.10Conclusion . . . 174

IV Learning 175

14 Introduction to Statistical Relational Learning 177
14.1 Introduction . . . 177
14.2 Inductive Logic Programming . . . 178
14.3 Statistical Relational Learning . . . 180
14.3.1 Parameter Learning . . . 182
14.3.2 Structure Learning . . . 183
14.4 Conclusion . . . 183

15 Distributed Learning in Probabilistic Logic Programming 185
15.1 Introduction . . . 185
15.2 Parameter Learning: EMBLEM . . . 186
15.3 Structure Learning: SLIPCOVER . . . 187
15.4 Distributed Parameter Learning: EMBLEM$^{\mathrm{MR}}$ . . . 188
15.5 Distributed Structure Learning: SEMPRE . . . 189
15.6 Experiments . . . 192
15.7 Conclusions . . . 196

16 Parameter Learning in Probabilistic Description Logics 197
16.1 Introduction . . . 197
16.2 EDGE . . . 197
16.2.1 Expectation Computation . . . 198
16.2.2 EDGE's Algorithm . . . 200
16.2.3 How to Use EDGE . . . 202
16.3 Conclusion . . . 203

17 Distributed Parameter Learning for Probabilistic Description Logics 205
17.1 Introduction . . . 205
17.2 Distributed Parameter Learning: EDGE$^{\mathrm{MR}}$ . . . 205
17.2.1 MapReduce View . . . 206
17.2.2 Scheduling Techniques . . . 207

17.2.3 EDGE^{MR}'s Algorithm 207
17.3 Experiments 211
17.4 Conclusions 213

18 Structure Learning in Probabilistic Description Logics 215
18.1 Introduction 215
18.2 The Learning Problem 216
18.3 Refinement Operators in Description Logics 216
18.4 CELOE 219
18.5 DL-Learner 221
18.6 Structure Learning: LEAP 222
18.6.1 Architecture 222
18.6.2 Interfacing CELOE and EDGE 223
18.6.3 LEAP 223
18.7 Related Work 226
18.8 Experiments 227
18.9 Conclusions 228

19 Distributed Stucture Learning in Probabilistic Description Logics 229
19.1 Distributed Structure Learning: LEAP^{MR} 229
19.2 Experiments 230
19.3 Conclusion 231

V Conclusions and Future Work 233

20 Conclusions 235

21 Future Work 239
21.1 Future Work on Inference 239
21.2 Future Work on Learning 240

Bibliography 243

Appendix 263

A List of Publications 263

List of Figures

1.1 Linked Open Data Cloud 4

3.1 Chapter dependency graph 14

4.1 Prolog SLD resolution tree 24

7.1 The Semantic Web Stack. 58
7.2 OWL 1 sublanguages. 62
7.3 OWL 2 sublanguages. 63

8.1 Some Pellet tableau expansion rules 70
8.2 HST Example 76
8.3 HST for All-MinAs$(Q, \mathcal{K})$ 78
8.4 Tableau expansion rules for building a pinpointing formula 84

9.1 Bayesian Network representing the dependency between $A(i)$ and $B(i)$. 91
9.2 Bayesian Network modeling the distribution over $A(i)$, $B(i)$, X_1, X_2, X_3. 92

11.1 MDD corresponding to Equation (11.2). 108
11.2 BDD for Example 11.5.1 equivalent to the MDD in Figure 11.1. 110
11.3 BDD for Example 11.5.2. 111
11.4 BDD for Example 11.5.3. 112
11.5 BDD for Example 11.5.4 with order $X_1 \prec X_2 \prec X_3 \prec X_4$. 113
11.6 BDD for Example 11.5.4 with order $X_1 \prec X_4 \prec X_2 \prec X_3$. 114

12.1 Bayesian network for a drug study domain. 122
12.2 Mutilated version of the Bayesian network of Figure 12.1 for computing the effect of a drug. 122
12.3 Architecture of cplint for causal inference. 128
12.4 LPAD for viral marketing. 130
12.5 Social network for the viral marketing example. 130
12.6 Average time for conditional and causal queries with 2 evidence literals. 131
12.7 Average time for conditional and causal queries with 4 evidence literals. 133
12.8 Average time for conditional and causal queries with 6 evidence literals. 133
12.9 Average time for conditional and causal queries with 8 evidence literals. 134
12.10`cplint` on SWISH interface. 145
12.11Graphical representations for query `pandemic` in Example 12.5.1. . . . 147
12.12Density of `X` of `mix(X)` obtained by the query in Example 12.5.2. . . . 148

12.13Prior and posterior densities of the argument `Y` of `value(0,Y)` obtained by likelihood weighting and particle filtering in Example 12.5.3 150
12.14Representation of the distributions in Example 12.5.4 151
12.15ProbLog program for viral marketing. 152

13.1 Example of ABox in TRILL 159
13.2 Code of the predicate `safe/3` 160
13.3 Code of the $\rightarrow$ *unfold* rule 161
13.4 Code of the $\rightarrow \sqcup$ rule 161
13.5 Application of expansions rules in TRILL 163
13.6 Code of the predicates `compute_prob/2` and `build_bdd/3`. 164
13.7 Predicates `test/2` and `build_f/3` 166
13.8 TRILL on SWISH interface. 167
13.9 Axioms from BRCA 169
13.10Axioms from DBPedia 169
13.11Axioms from Biopax 171
13.12Axioms from Vicodi 172

15.1 SEMPRE speedup graph 193

16.1 CBDD equivalent to the BDD in Example 11.5.3 199

17.1 Scheduling techniques of EDGE^{MR}. 208
17.2 Speedup of EDGE^{MR} 212
17.3 Memory consumption of EDGE^{MR} 213

18.1 Illustration of a search tree in a top down refinement approach. 220
18.2 Positive and negative examples in a class learning problem 220
18.3 The architecture of DL-Learner (redrawing from [244]). 222
18.4 LEAP Architecture 223

19.1 LEAP^{MR} architecture. 229
19.2 Speedup of LEAP^{MR} 231

List of Tables

5.1 M_{DB_1} for the finite program DB_1. 33
5.2 P_{F_1} and P_{DB_1} for the finite program DB_1. 33

7.1 Some DL constructors with their associated DL language symbols. . . . 49
7.2 Syntax and semantics of common concept and individual constructors. 51
7.3 Syntax and semantics of common role constructors. 51
7.4 Syntax and semantics of common datatype and data value constructors 52
7.5 Correspondence between DL axioms and their translation into FOL . . 57
7.6 Terminology comparison of FOL, DL, and OWL. 59
7.7 DL Axiom Woman $\sqsubseteq$ Person in different OWL 2 syntaxes. 60
7.8 Most common OWL expressions in DL and Manchester OWL syntax. . 61
7.9 Most common OWL axioms in DL and Manchester OWL syntax. . . . 61

12.1 Execution time for conditional and causal queries with 2 evidence literals 132
12.2 Execution time for conditional and causal queries with 4 evidence literals 132
12.3 Execution time for conditional and causal queries with 6 evidence literals 134
12.4 Execution time for conditional and causal queries with 8 evidence literals. 135
12.5 Mean Squared Error for approximate causal inference 135

13.1 Average time of BUNDLE, TRILL and TRILL^P for different datasets. . 170
13.2 Average time of BUNDLE, TRILL and TRILL^P for synthetic datasets . 170

15.1 SEMPRE execution time 192
15.2 AUC-PR and execution time (in seconds) on the Mutagenesis dataset . 194
15.3 AUC-PR and execution time (in seconds) on the Carcinogenesis dataset 194
15.4 AUC-PR and execution time (in seconds) on the IMDB dataset 195
15.5 AUC-PR and execution time (in seconds) on the HIV dataset 195

17.1 Characteristics of the datasets used for evaluation. 211
17.2 Comparison between EDGE and EDGE$^{\mathrm{MR}}$ 212

18.1 Results of the experiments in terms of AUCPR and AUCROC averaged over the folds with EDGE and LEAP 228

List of Algorithms

8.1 Tableau algorithm executed by Pellet. 69
8.2 SingleMinA algorithm. 73
8.3 Black-Box pruning algorithm. 74
8.4 Hitting Set Tree Algorithm. 79
11.1 Splitting Algorithm. 104
11.2 Function Prob 110
12.1 Algorithm for computing the conditional probabilities. 119
12.2 Algorithm for preparing the knowledge base for exact causal inference. 126
12.3 Algorithm for preparing the knowledge base for approximate causal inference. 127
13.1 Function Bundle: computation of the probability of a query Q given the (probabilistic) KB $\mathcal{K}$. 157
15.1 EMBLEM algorithm. 186
15.2 Function SLIPCOVER 188
15.3 Function EMBLEM$^{\text{MR}}$ 189
15.4 Function SEMPRE 191
16.1 EDGE 201
16.2 Expectation 201
16.3 Maximization 202
16.4 GetForward 202
16.5 GetBackward 203
17.1 Function EDGE$^{\text{MR}}$ 210
18.1 LEAP Algorithm 225
19.1 LEAP$^{\text{MR}}$ 230

List of Acronyms

BDD Binary Decision Diagram

CNF Conjunctive Normal Form

CRV continuous random variable

DL Description Logic

DNF Disjunctive Normal Form

EM Expectation-Maximization

FOL First-Order Logic

HST hitting set tree

ICL Independent Choice Logic

ILP Inductive Logic Programming

KB knowledge base

LP Logic Programming. Logic Program

LPAD Logic Program with Annotated Disjunctions

MCMC Markov Chain Monte Carlo

MDD Multivalued Decision Diagram

mgu most general unifier

MPI Message Passing Interface

OWL Web Ontology Language

PDL Probabilistic Description Logic

PILP Probabilistic Inductive Logic Programming

PLP Probabilistic Logic Programming. Probabilistic Logic Program

SRL Statistical Relational Learning

Part I

Introduction

Chapter 1

Motivation

In the last few years we have witnessed the spread of the Semantic Web and its technologies for knowledge representation and reasoning. The Semantic Web is promoted by the World Wide Web Consortium and encourages the publication on the Internet of particular information, called *semantic content*, that can be processed and understood by machines.

In 2013, it was estimated that web content reached 4 Zettabytes. This information indicates how important it is to develop tools capable of representing and analyzing a huge amounts of information, which are no longer manageable by traditional (relational) databases.

Since the dawn of artificial intelligence, the problem of knowledge representation has been one of the most addressed problems. Logic programming is an area of research in which knowledge is represented by formalisms based on First-Order Logic (FOL).

Another approach to represent knowledge is by means of ontologies. An ontology is a formal and explicit representation of a domain of interest, which solves the problems of ambiguity between entities. An ontology can be expressed with a description logic.

Description Logics (DLs) are a family of logical formalisms used to model knowledge in terms of concepts, roles and properties. They are usually a decidable fragment of FOL.

Different ontologies may have a common concept. For example, the concept *Eye* within an ontology concerning human pathologies is the same in an ontology of human anatomy. The fact that an entity is the same in two different knowledge bases (KBs) establishes a sort of *semantic* link. Linked Open Data is a methodology for publishing Open Data (structured data without copyright and automatically processable) on the Web, so as to link data and make it easier for both man and machine to access additional information. With this methodology we can link KBs that come from different areas of expertise, generating a huge *semantic* network called Linked Open Data Cloud, shown in Figure 1.1, in which each node is a KB and each link between nodes means there are some relationships between entities from different KBs (i.e. owl:sameAs).

Logic programming languages and description logics are logical formalisms that can be used to represent datasets containing huge amount of information. However, in the real world domain, information may be uncertain, and neither logic programs nor description logics can represent it. Therefore, it becomes essential to provide methods for representing this type of information.

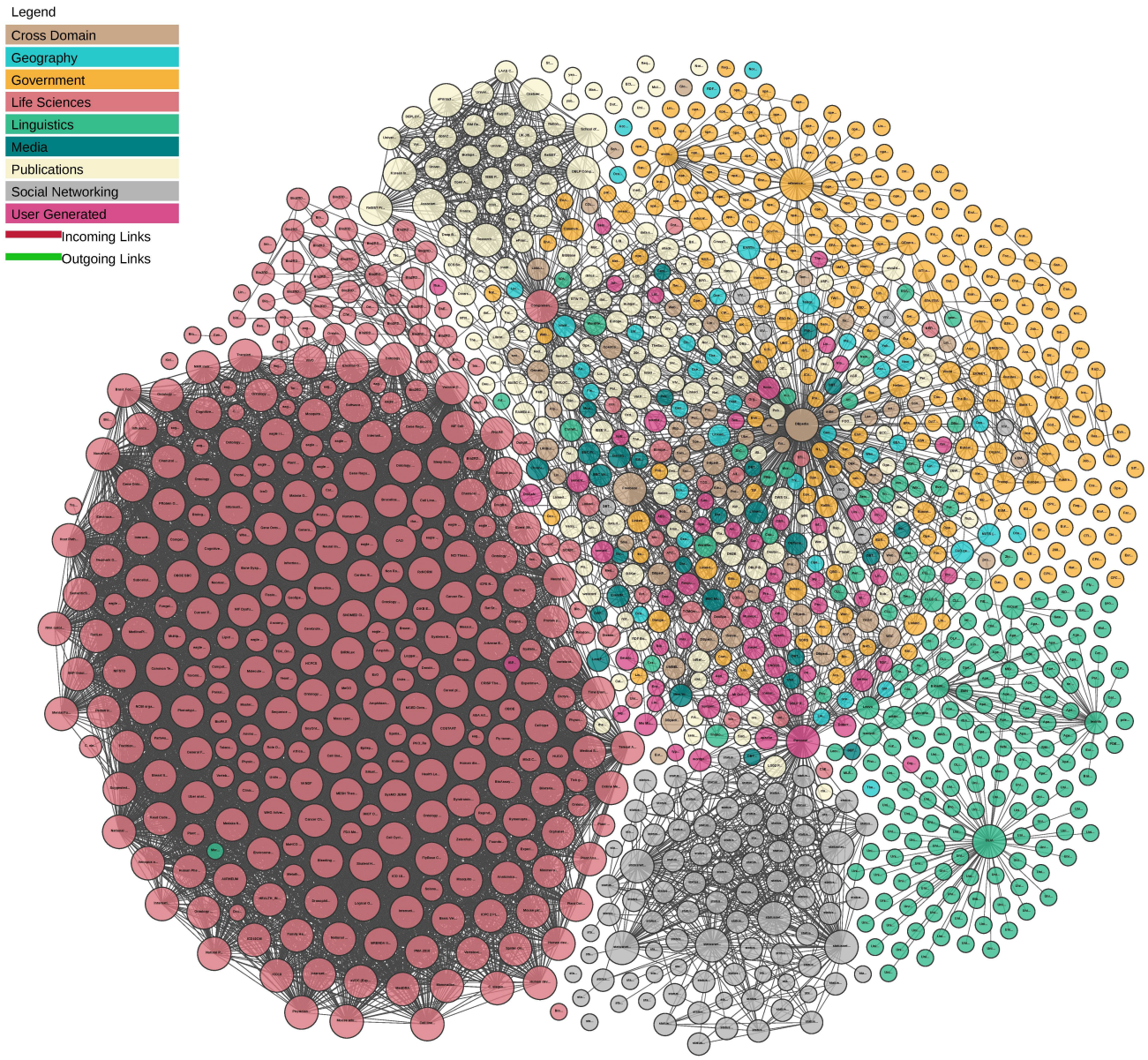


Figure 1.1: Linked Open Data Cloud. *Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak.* `http://lod-cloud.net/`. Last updated: 2017-08-22.

In 1995 Taisuke Sato defined the *distribution semantics* [1], which provides a powerful mechanism to combine logic with probability theory, in fact this semantics yielded a new subfield of SRL called Probabilistic Logic Programming (PLP) and several PLP languages, such as PRISM, ProbLog and LPADs.

In order to apply the acquired knowledge in the field of Probabilistic Logic Programming in the case of Description Logics, in [2, 3] the authors formalized DISPONTE, a semantics that applies the distribution semantics to description logics, thus defining Probabilistic Description Logics (PDLs).

Information can become knowledge only if it has some useful and even applicable use. Thus, once we defined these logical formalisms and their semantics, a first important step is to define and develop systems for automatic reasoning, both for PLP languages and PDLs. In fact, while in traditional databases all the knowledge is explicit, a knowledge base expressed by a logical formalism contains implicit knowledge that can be made explicit only by using an inference system, also called *reasoner*.

Reasoners, however, do not introduce new knowledge, but, as just said, they just make explicit an already acquired knowledge. Therefore, the next step is to develop automatic methods to enrich an initial knowledge base with new knowledge. The field of artificial intelligence that concerns the development of this type of methods is known under the name of *Machine Learning*.

Statistical Relational Learning (SRL) is a relatively new research field that combines principles and ideas from three important areas of Artificial Intelligence: machine learning, knowledge representation and reasoning on uncertainty. In SRL the knowledge is represented with a probabilistic logical formalism. The development of systems in the field of SRL is of considerable importance, to solve the problems arising from this discipline.

An important problem faced by machine learning researchers is managing Big Data. Learning from big data raises a lot of problems. One of the most common approaches to solving them is to use distributed algorithms.

Chapter 2

Aims of the Thesis

The main objective of this thesis is to propose approaches for reasoning, parameter learning and structure learning on knowledge bases represented with a probabilistic logic-based formalism.

We tested the proposed approaches on real-world datasets and their performances was comparable or superior to other state-of-the-art systems.

The first part of this thesis concerns reasoning over uncertain data. In particular, with regard to reasoning in PLP, we present the latest advances in the `cplint` system, which allows hybrid programs, i.e. programs where some of the random variables are continuous, and causal inference. Moreover we developed a web interface for `cplint`, named `cplint` on SWISH, which allows the user to experiment with the system without having to install anything on the local machine. To perform inference on PDLs that follow DISPONTE, a suite of algorithms is presented: BUNDLE ("Binary decision diagrams for Uncertain reasoNing on Description Logic thEories"), TRILL ("Tableau Reasoner for descrIption Logics in Prolog" and TRILL^P ("Tableau Reasoner for descrIption Logics in Prolog powered by Pinpointing formula").

The second part of our thesis focuses on machine learning. Machine learning is a huge area of research, with several subfields and algorithms. We can group machine learning algorithms into *supervised*, *unsupervised*, and *reinforcement learning* methods.

- In **supervised learning**, training data is composed by examples where each of them is labeled with the correct output class. Given this training data, the aim of the learning algorithm is to find a function mapping between input and output.
- In **unsupervised learning**, training data is not labeled and the aim of the learner is to extract relationships or correlations from the available data.
- In **reinforcement learning**, the learning algorithm performs actions in an environment and receives rewards or punishments as feedback. Given a certain state of the environment, the learner tries to maximize the rewards and minimize the punishments by choosing the appropriate actions to perform.

Our research area is inside the Statistical Relational Learning (SRL) field that can be seen as a subfield of machine learning. All the learning systems proposed in this work are supervised. Moreover we use probabilistic logic-based formalisms to represent knowledge. Logical formalisms fall into the category of symbolic formalisms (as

opposed to non-symbolic and sub-symbolic). This means that our learning approaches produce logical, human understandable solutions. Non-symbolic methods such as neural networks do not produce human understandable results. Sub-symbolic methods are between these two extremes.

SRL is a relatively young field. There are many opportunities to develop new methods for real-world problems. In SRL the two main learning problems are:

- *parameter learning*, given the probabilistic knowledge base (composed of logic formulas) we want to learn the parameters, and
- *structure learning*, we want to learn both new probabilistic logic formulas (the structure) and the parameters.

In this dissertation we propose systems to solve both these problems, whether we use PLP languages or PDLs that follow DISPONTE.

The learning systems proposed in these thesis can be grouped into two categories: systems for PLP and systems for PDLs. In the past, various learning algorithms for PLP have been proposed, among these we mention EMBLEM [4] for parameter learning and SLIPCOVER [5] for structure learning. However, the execution of these algorithms is rather expensive from a computational point of view, taking a few hours on datasets of the order of MBs. In order to efficiently manage larger datasets in the era of Big Data and Linked Open Data, it is of foremost importance to develop algorithms and techniques for improving the performances and reaching scalability. One solution is to distribute the algorithms using modern computing infrastructures such as clusters and clouds. We thus developed the distributed versions of EMBLEM and SLIPCOVER: EMBLEM$^{\text{MR}}$ and SEMPRE ("Structure lEarning by MaPREduce").

As learning systems for PDLs, we present EDGE ("Em over bDds for description loGics paramEter learning") for parameter learning and LEAP ("LEArning Probabilistic description logics") for structure learning. For the same reasons that led us to develop SEMPRE, we extended EDGE and LEAP by developing their distributed versions: EDGE$^{\text{MR}}$ and LEAP$^{\text{MR}}$.

Chapter 3

Structure of the Thesis

3.1 Structure

The content of this thesis is divided into individual parts and chapters as follows.

Part I - Introduction
This part contains introductory chapters explaining the motivations (Chapter 1), aims of this thesis (Chapter 2) and the structure of the thesis (this chapter).

Part II - Probabilistic Logics
Most logical formalisms are based on First-Order Logic (FOL), Part II provides the background knowledge to understand them and the main differences between Logic Programming (LP) and FOL. Chapter 5 presents the distribution semantics as defined by Taisuke Sato. In Chapter 6 we illustrate LPADs and ProbLog, two of the most famous PLP languages, and how they apply the distribution semantics. Besides logic programming languages we can represent knowledge by means of Semantic Web technology. Web Ontology Language (OWL) is one of them and it based on Description Logics. Description Logics and OWL are both discussed in Chapter 7. Like in the LP framework, DLs are not capable of representing uncertain information. DISPONTE, explained in Chapter 9, applies the distribution semantics to description logics and we named the resulting formalism Probabilistic Description Logics (PDLs).

Part III - Inference in Probabilistic Logics
In this part of the thesis we introduce systems for probabilistic logical inference. Binary Decision Diagrams (BDDs), illustrated in Chapter 10 are a graphic method to represent Boolean formulas. They are used when performing exact probabilistic logical inference, see Chapter 11. In Chapter 12 and Chapter 13 we present inference systems for LPADs and PDLs respectively.

Part IV - Learning
This part of the thesis concerns machine learning, in particular one of its subfields known as Statistical Relational Learning (SRL). An introduction to SRL and a discussion of its main learning problems are provided in Chapter 14. The main learning problem are parameter learning and structure learning. In Chapter 15 we propose a

distributed structure learning algorithm for LPADs called SEMPRE. For parameter learning in the PDL paradigm we illustrate EDGE in Chapter 16, and we present its distributed version EDGE^{MR} in Chapter 17. For structure learning, instead, we propose LEAP in Chapter 18 and LEAP^{MR}, which integrates EDGE^{MR} into LEAP, in Chapter 19.

Part V - Conclusions and Future Work
This part closes this thesis with some final remarks on our study (Chapter 20), including a discussion of the limitations of our systems and possible future directions (Chapter 21).

3.2 Thesis Contributions

We can classify the contributions of this thesis into four categories. The first two are related to probabilistic logical inference, whereas the last two to Statistical Relational Learning (SRL).

3.2.1 Inference in Probabilistic Logic Programming

In Chapter 12 we present the latest advances in the `cplint` system. `cplint` contains programs for exact and approximate probabilistic logical inference. We extended this system in order to perform causal inference and reason over hybrid probabilistic logic programs, i.e. programs where some of the random variables are continuous. Moreover in this chapter we propose `cplint` on SWISH, a web application that let the user to experiment with `cplint` without having to install anything on the local machine.

Chapter 12 is based on the following publications:

- F. Riguzzi, G. Cota, E. Bellodi, and R. Zese. "Causal inference in cplint". In: *International Journal of Approximate Reasoning* 91 (2017), pp. 216–232. ISSN: 0888-613X. DOI: `https://doi.org/10.1016/j.ijar.2017.09.007`. URL: `https://www.sciencedirect.com/science/article/pii/S0888613X17301640`.

- M. Alberti, E. Bellodi, G. Cota, F. Riguzzi, and R. Zese. "`cplint` on SWISH: Probabilistic Logical Inference with a Web Browser". In: *Intelligenza Artificiale* 11.1 (2017), pp. 47–64. DOI: `10.3233/IA-170105`

- F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. "Probabilistic Logic Programming on the Web". In: *Software: Practice and Experience* 46.10 (Oct. 2016), pp. 1381–1396. DOI: `10.1002/spe.2386`

In all these works I have been involved in the development of the extensions of the cplint system and its web application cplint on SWISH. In addition, with regard to the causal inference of cplint, I have taken care of all the experiments, which are easily reproducible, by downloading the scripts at `https://goo.gl/wq3X9i`. In order to spread the use of cplint, I also have developed an online tutorial [9] available at `http://ds.ing.unife.it/~gcota/plptutorial/`.

3.2.2 Inference in Probabilistic Description Logics

DISPONTE is a semantics for Probabilistic Description Logics (PDLs) defined in [3]. This semantics assumes that all the axioms in the knowledge base are independent, i.e. it makes the so-called independence assumption. In Subsection 9.2.3 we prove that this assumption is valid. This subsection is based on the publication:

- R. Zese, E. Bellodi, F. Riguzzi, G. Cota, and E. Lamma. "Tableau Reasoning for Description Logics and its Extension to Probabilities". In: *Annals of Mathematics and Artificial Intelligence* (2016), pp. 1–30. DOI: 10.1007/s10472-016-9529-3. URL: http://dx.doi.org/10.1007/s10472-016-9529-3f

In Chapter 13 we illustrate three PDL inference systems: BUNDLE [11, 3], TRILL and TRILL^P. Moreover we present a web interface for TRILL and TRILL^P, called TRILL on SWISH. This chapter is based on the following publications:

- R. Zese, E. Bellodi, F. Riguzzi, G. Cota, and E. Lamma. "Tableau Reasoning for Description Logics and its Extension to Probabilities". In: *Annals of Mathematics and Artificial Intelligence* (2016), pp. 1–30. DOI: 10.1007/s10472-016-9529-3. URL: http://dx.doi.org/10.1007/s10472-016-9529-3f
- E. Bellodi, E. Lamma, F. Riguzzi, R. Zese, and G. Cota. "A web system for reasoning with probabilistic OWL". in: *Software: Practice and Experience* 47.1 (2017), pp. 125–142

In all these works, in addition to writing the papers, I have dealt with part of the implementation of the systems mentioned above. In addition, much of the work done in the field of inference in PDLs consists of "hidden" engineering contributions on previously developed systems, which have not led to publications. For example, I worked on the development of the later versions of BUNDLE, which has undergone several modifications and optimizations, in order to be efficiently used by learning algorithms. I also focused on the integration of BUNDLE into DL-Learner, increasing the chances for it to be used by others.

3.2.3 Learning Systems in Probabilistic Logic Programming

In SRL the two main learning problems are parameter learning and structure learning. For the former problem, in [4] the authors developed EMBLEM, a parameter learning algorithm for LPADs, which are a PLP language. Then, to solve the latter, the same authors, in [5] developed a structure learning algorithm called SLIPCOVER.

In Chapter 15 we present the distributed versions of these algorithms called EMBLEM$^{\text{MR}}$ and SEMPRE, respectively. This chapter is based on the following publication:

- F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. "Scaling Structure Learning of Probabilistic Logic Programs by MapReduce". In: *Proceedings of the 22nd European Conference on Artificial Intelligence*. Ed. by M. Fox and G. Kaminka. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 1602–1603. DOI: 10.3233/978-1-61499-672-9-1602

In this work I have contributed to the implementation of the mentioned distributed systems.

3.2.4 Learning Systems in Probabilistic Description Logics

EDGE [14] is a parameter learning algorithm for PDLs. In Chapter 17 we propose EDGE^{MR} which is a distributed version of EDGE. This chapter is based on the publication:

- G. Cota, R. Zese, E. Bellodi, F. Riguzzi, and E. Lamma. "Distributed Parameter Learning for Probabilistic Ontologies". In: *25th International Conference on Inductive Logic Programming*. Ed. by K. Inoue, H. Ohwada, and A. Yamamoto. 2015

In Chapter 18 we propose a structure learning algorithm for PDLs, called LEAP. In Chapter 19 we present its distributed version called LEAP^{MR}. These chapters are based on the following publications:

- F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. "Learning Probabilistic Description Logics". English. In: *Uncertainty Reasoning for the Semantic Web III*. ed. by F. Bobillo et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer International Publishing, 2014, pp. 63–78. ISBN: 978-3-319-13412-3. DOI: 10.1007/978-3-319-13413-0_4

- G. Cota, R. Zese, E. Bellodi, E. Lamma, and F. Riguzzi. "Structure Learning with Distributed Parameter Learning for Probabilistic Ontologies". In: *Doctoral Consortium of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2015)*. Ed. by J. Hollmen and P. Papapetrou. 2015, pp. 75–84. ISBN: 978-952-60-6443-7. URL: http://urn.fi/URN:ISBN:978-952-60-6443-7

Work on learning systems for PDLs was maybe the topic on which I have concentrated most of my efforts. I was the main developer of these systems and my contributions ranged from theoretical contributions to the actual implementation of these systems. In addition, for all these systems I contributed to the experimental phase.

Here too, there were many "hidden" engineering contributions. I am not an author of the paper on EDGE [14], but I have optimized it several times in order to be used with LEAP. In addition, I worked on the integration of EDGE into DL-Learner.

3.3 How to read this thesis

The aim of this work is to propose systems for inference and learning on probabilistic logics. In this thesis we use two logical formalisms for representing uncertain information: Logic Programs with Annotated Disjuntions (LPADs) and Probabilistic Description Logics (PDLs). LPADs are a Probabilistic Logic Programming (PLP) language, whereas PDLs are based on Description Logics (DLs) and follow the DISPONTE semantics (see Chapter 9). Therefore the proposed systems can be split into two main categories: PLP systems and PDL systems.

This thesis is designed such that can be read by both experts of logic programming and experts of description logics. If the reader is only interested in PLP systems there is no need to read the whole thesis but only a part of it. The same if the reader is

only interested in PDL systems. Figure 3.1 depicts the dependencies among different chapters. The common chapters useful to understand both PLP and PDL systems are in gray; the chapters concerning PLP systems are in blue; whereas the chapters dealing with PDLs that follow DISPONTE and the systems proposed for this formalism are in green.

We define the following reading sequences of the main chapters essential to understand this thesis according to reader's interest:

- **Chapter sequence for PLP systems:**
  - **Part II - Probabilistic Logics:** 4, 5 and 6.
  - **Part III - Inference in Probabilistic Logics:** 10, 11 and 12.
  - **Part IV - Learning:** 14 and 15.
- **Chapter sequence for PDL systems:**
  - **Part II - Probabilistic Logics:** 4, 5, 7, 8 and 9.
  - **Part III - Inference in Probabilistic Logics:** 10, 11 and 13.
  - **Part IV - Learning:** 14, 16, 17, 18 and 19.

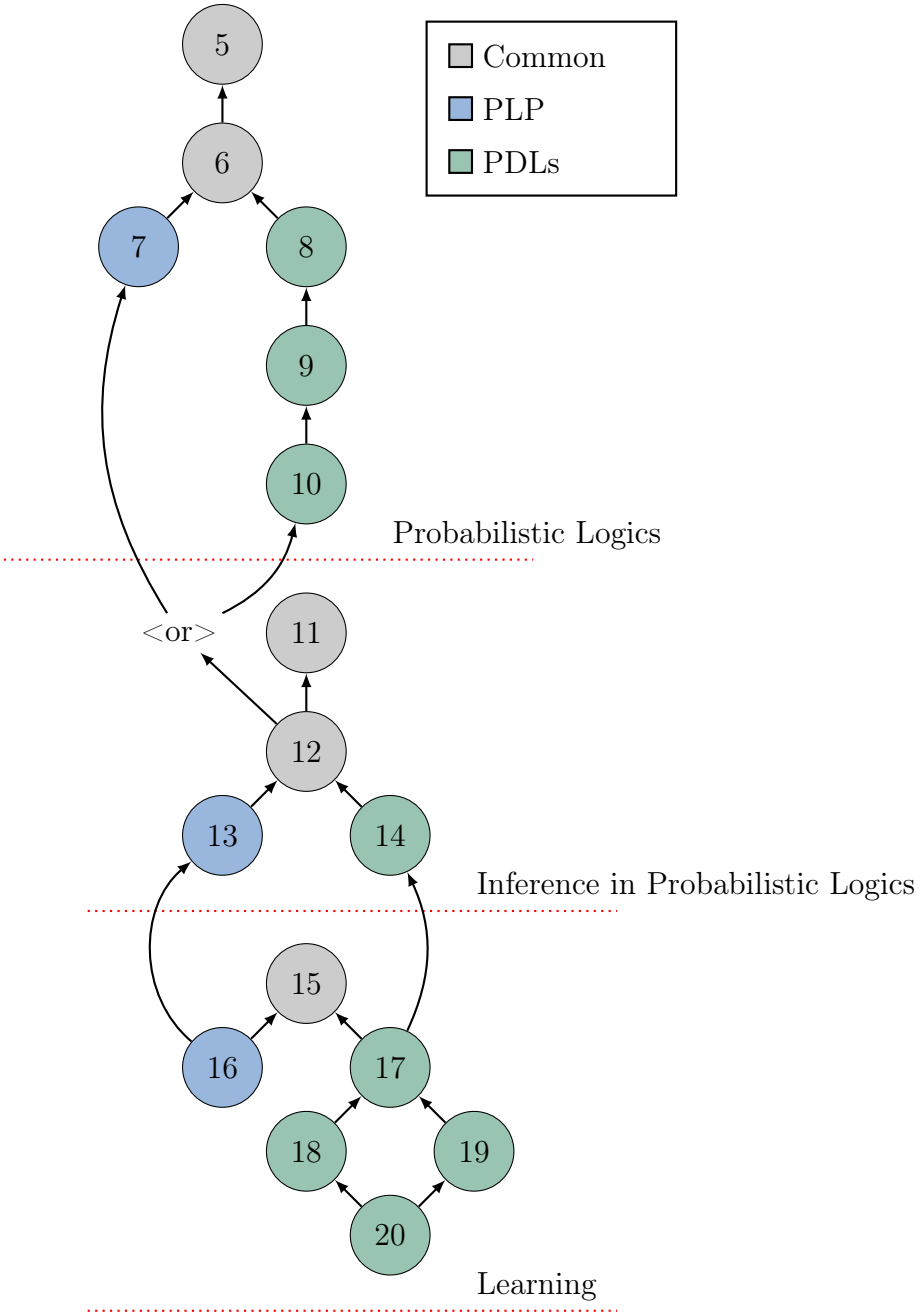


Figure 3.1: Dependency graph of the main chapters. For instance, to understand Chapter 17 you have to read chapters 14 and 15 first.

Part II

Probabilistic Logics

Chapter 4

Fundamentals of First-Order Logic and Logic Programming

This chapter introduces the main concepts of First-Order Logic (FOL) and Logic Programming (LP) and it underlines the main differences between FOL and LP.

The chapter is organized as follows. After a brief introduction in Section 4.1, Sections 4.2 and 4.3 provide the basis for understanding First-Order Logic and Logic Programming respectively. Section 4.4 compares these two frameworks. The final section (Section 4.5) draws some final considerations.

4.1 Introduction

Logic proved to be a very powerful tool for representing the complexity of real world domains, where the entities of interest are composed of subparts connected by a network of relations. Probabilistic logic tries to combine logic with probability theory. In this thesis we mainly use two family of probabilistic logical formalisms for representing knowledge: Probabilistic Logic Programming (PLP) languages, see Chapter 6 and Probabilistic Description Logics (PDLs), see Chapter 9. Before illustrating them, we overview the fundamentals of First-Order Logic (FOL) and Logic Programming (LP). Indeed PLP languages are extensions of logic programming languages (usually Prolog), whereas many DLs can be seen as decidable fragments of FOL.

4.2 First-Order Logic

First-Order Logic (FOL), also known as **predicate logic**, extends propositional logic by allowing the use of formulas that contain variables.

4.2.1 Syntax

FOL is a formal system with an alphabet Σ consisting of seven classes of symbols:

- **Logical Variables**, a sequence of alphanumeric characters that refer to objects in the domain.

  - e.g. X, Y, Z

- **Constants**, objects in the domain.
  - e.g. $a, b, c, \mathit{jeff}, \mathit{ann}$

- **Function symbols**, a relation that maps n objects to another object, with arity (number of arguments) $n > 0$. We use the notation f/n to denote a function f with arity n.
  - e.g. if we have these formulas $f(a, b)$, $s(X)$, *mother_ of(mary)*, the functions are $f/2$, $s/1$, $\mathit{mother\_of}/1$.

- **Predicate symbols**, a relations that maps n objects onto truth values, with arity $n > 0$. As for functions, we use the notation p/n to denote a predicate p with arity n
  - e.g. if we have these formulas $p(X)$, $\mathit{mother}(\mathit{ann}, \mathit{mary})$, $\mathit{parent}(\mathit{jeff}, \mathit{paul})$, the predicates are $p/1$, $\mathit{mother}/2$, $\mathit{parent}/2$.

- **Logical connectives**, used to connect formulas.
  - e.g. $\neg, \wedge, \vee, \leftarrow, \leftrightarrow$

- **Quantifiers**, expressing generality.
  - e.g. $\forall$ universal quantifier ("for all"), $\exists$ existential quantifier ("for some", "there exists")

- **Punctuation symbols**, used to make formulas more readable.
  - e.g. ')', '('

A **signature**[1] consists of a triple $\Lambda = \langle C, F, P \rangle$, where C, F and P are sets of constant, function and predicate symbols respectively. A **term** is a variable, a constant, or a function applied to terms. An **atom** is a predicate symbol followed by its terms, e.g. $\mathit{parent}(\mathit{jeff}, \mathit{paul})$. A **literal** is an atom (*positive literal*) or the negation of an atom (*negative literal*), e.g. $\mathit{mother}(\mathit{ann}, \mathit{mary}), \neg \mathit{mother}(\mathit{mary}, \mathit{ann})$. FOL formulas are recursively constructed from atoms using logical connectives and quantifiers. If ψ and ϕ are formulas then the following are formulas too:

- $\neg\psi$ (*logical negation*), which is true $\Leftrightarrow$ ψ is not true.
- $\psi \wedge \phi$ (*logical conjunction*), which is true $\Leftrightarrow$ both ψ and ϕ are true.
- $\psi \vee \phi$ (*logical disjunction*), which is true $\Leftrightarrow$ ψ or ϕ is true.
- $\psi \leftarrow \phi$ (*logical implication*), which is true $\Leftrightarrow$ ϕ is false or ψ is true.
- $\psi \leftrightarrow \phi$ (*logical equivalence*), is a shorthand for $(\psi \leftarrow \phi) \wedge (\phi \leftarrow \psi)$

[1] Sometimes called *vocabulary*.

- $\exists X\psi$ (*existential quantification*), which is true $\Leftrightarrow$ ψ is true for at least one object in the domain replaced for X.
- $\forall X\psi$ (*universal quantification*), which is true $\Leftrightarrow$ ψ is true for every object replaced for X.

An occurrence of a variable is *free* iff it is outside the scope of a quantifier of that variable, otherwise it is *bound*. For instance:

- in the formula $\forall X p(X, Y)$, X is bound, whereas Y is free;
- in the formula $q(X) \vee \exists p(X)$, the first occurrence of X is free, whereas the second one is bound.

A formula which contains at least one free occurrence of a variable is called **open formula**, otherwise is called **closed formula** or **sentence**.

A **clause** is a formula where all the variables are universally quantified and is of the form

$$A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_m \tag{4.1}$$

where A_i and B_j are atoms. Formula 4.1 is logically equivalent to:

$$A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m \tag{4.2}$$

$A_1 \vee \cdots \vee A_n$ is the head of the clause, whereas $B_1 \wedge \cdots \wedge B_m$ is the body. If the body is empty the clause is called **fact**[2]. A clause can be read as "if the conjunction of all the B_js are true, the disjunction of all A_is are true".

An **expression** is a term, atom, conjunction or clause. An expression is called *ground* if it does not contain variables. A FOL **theory** is a set of formulas that implicitly form a conjunction. **Clausal logic** is an important subset of FOL. A **clausal theory** consists of a set of clauses.

The **Herbrand universe** $hu(T)$ of a theory T is the set of all the ground terms that can be built from functions and constants appearing in T. The **Herbrand base** of a FOL theory T, denoted as $hb(T)$, is the set of all the ground atoms constructed with the predicates in the alphabet of T and the terms of the Herbrand universe.

Example 4.2.1
Given the following FOL theory T

$$\begin{aligned} &parent(jeff, paul)\\ &parent(paul, ann)\\ &grandparent(X, Y) \leftarrow parent(X, Z), parent(Z, Y) \end{aligned}$$

its Herbrand universe $hu(T)$ *is*

$$hb(T) = \{jeff, paul, ann\}$$

whereas, its Herbrand base $hb(T)$ *is*

$$\begin{aligned} hb(T) = \{&parent(jeff, jeff), parent(paul, paul), parent(ann, ann),\\ &parent(jeff, paul), parent(jeff, ann), parent(paul, jeff), \ldots,\\ &grandparent(jeff, jeff), grandparent(paul, paul), \ldots\} \end{aligned}$$

[2] A **fact** can also be seen as a rule that has *true* as its body.

4.2.2 Tarski's semantics

Usually when we talk about the semantics of FOL we refer to Tarski's semantics, defined by the Polish logician Alfred Tarski.

Tarski's semantics can be defined as a **structure** consisting of a triple $\mathbf{S} = \langle U, \Lambda, \mathcal{I} \rangle$, where U is a non-empty set, called *domain* or *universe*, which defines the domain of discourse, Λ is a signature, and $\mathcal{I}$, called *interpretation*, is a function which assigns a "meaning" to every symbol in the signature Λ.

Definition 4.1 FOL Interpretation
An interpretation $\mathcal{I}$ indicates how a signature Λ is interpreted on a domain U. An **interpretation** $\mathcal{I}$ of a signature Λ assigns meanings as follows.

- For each constant symbol c in Λ, $\mathcal{I}(c)$ assigns an individual $c^{\mathcal{I}} \in U$.
- For each function symbol f of arity n, $\mathcal{I}(f)$ assigns a function $f^{\mathcal{I}} : U^n \to U$.
- For each predicate symbol p of arity n, $\mathcal{I}(p)$ assigns a relation $p^{\mathcal{I}}$ over U^n, i.e. $p^{\mathcal{I}} \subseteq U^n$.

□

Definition 4.2 FOL Assignment Function
An **assignment** ν associates each variable of a formal language $\mathcal{L}$ to an individual in the domain of discourse U. The assignment function is required in order to give a meaning to formulas with free variables. □

Combining interpretations and assignments provides a way to assign meanings to terms.

Definition 4.3 Interpretation given an Assignment
Let t be a term, $\mathbf{S}$ a structure $\mathbf{S} = \langle U, \Lambda, \mathcal{I} \rangle$, with domain U and an interpretation $\mathcal{I}$ of a signature Λ, and ν an assignment, we define the function $\mathcal{I}^{\nu}(t)$ (interpretation $\mathcal{I}$ given an assignment ν), that assigns a meaning to t, as follows

- if t is a constant, then $\mathcal{I}^{\nu}(t) = \mathcal{I}(t)$;
- if t is a variable, then $\mathcal{I}^{\nu}(t) = \nu(t)$;
- Given the terms $t_1, \ldots, t_n$, if t is a function $f(t_1, \ldots, t_n)$, then

$$\mathcal{I}^{\nu}(t) = \mathcal{I}^{\nu}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathcal{I}^{\nu}(t_1), \ldots, \mathcal{I}^{\nu}(t_n))$$

□

With the notation $\nu[x/v]$ we indicate a new assignment function which is equal to ν except for the variable x that is assigned to the individual $v \in U$. We provide now the definition of *satisfiability* of a logic formula according to Tarski's semantics.

Definition 4.4 Satisfiability of a formula with Tarski semantics
Given a structure $\mathbf{S} = \langle U, \Lambda, \mathcal{I} \rangle$, with domain U and an interpretation $\mathcal{I}$ of a signature Λ, and the assignment function ν, the satisfiability of a formula ψ, denoted as $\mathcal{I}^{\nu} \models \psi$, is defined as follows

- if ψ is an atom $\psi = p(t_1, \ldots, t_n)$, $\mathcal{I}^\nu \models \psi$ iff $p^\mathcal{I}(\nu(t_1), \ldots, \nu(t_n)) \in \mathcal{I}(p)$;
- if $\psi = \neg\phi$, $\mathcal{I}^\nu \models \psi$ iff it is not the case $\mathcal{I}^\nu \models \phi$, also written $\mathcal{I}^\nu \not\models \phi$;
- if $\psi = \phi \wedge \gamma$, $\mathcal{I}^\nu \models \psi$ iff $\mathcal{I}^\nu \models \phi$ and $\mathcal{I}^\nu \models \gamma$;
- if $\psi = \phi \vee \gamma$, $\mathcal{I}^\nu \models \psi$ iff $\mathcal{I}^\nu \models \phi$ or $\mathcal{I}^\nu \models \gamma$;
- if $\psi = \phi \rightarrow \gamma$, $\mathcal{I}^\nu \models \psi$ iff $\mathcal{I}^\nu \not\models \phi$ or $\mathcal{I}^\nu \models \gamma$;
- if $\psi = \phi \leftrightarrow \gamma$, $\mathcal{I}^\nu \models \psi$ iff $\mathcal{I}^\nu \models \phi$ and $\mathcal{I}^\nu \models \gamma$, or $\mathcal{I}^\nu \not\models \phi$ and $\mathcal{I}^\nu \not\models \gamma$;
- if $\psi = \exists X\phi$, $\mathcal{I}^\nu \models \psi$ iff, for some $v \in U$, $\mathcal{I}^{\nu[x/v]} \models \phi$;
- if $\psi = \forall X\phi$, $\mathcal{I}^\nu \models \psi$ iff, for all $v \in U$, $\mathcal{I}^{\nu[x/v]} \models \phi$.

In ψ is satisfied in an interpretation $\mathcal{I}$ given an assignment ν, we also say that a formula ψ is true in an interpretation $\mathcal{I}$ given an assignment ν. □

If ψ is a **sentence**, i.e. a formula without free variables, the satisfiability of a sentence ψ does not depend on ν, hence we can simply write $\mathcal{I} \models \psi$.

Definition 4.5 FOL Model
Let S be a set of sentences, we say that $\mathcal{I}$ *satisfies* S or is a **model** of S iff $\mathcal{I} \models \psi$ for all $\psi \in S$. □

Definition 4.6 Logical entailment
We say that the sentence ψ is a **logical entailment** or **logical consequence** of the set of sentences S, denoted as $S \models \psi$ if every model of S is also a model of ψ. In this case, we also say that S **entails** ψ, or ψ **is a consequence of** S, or again ψ **follows from** S. □

Definition 4.7 Herbrand Interpretation
A **Herbrand interpretation** $\mathcal{I}$ for a FOL theory T is an interpretation of a structure $\mathbf{S} = \langle U, \Lambda, \mathcal{I}\rangle$ whose domain U is the Herbrand universe of T, i.e. $U = hu(T)$. □

Definition 4.8 Herbrand Model
A Herbrand interpretation is a **Herbrand model** of a theory if it satisfies all formulas in the theory, i.e. all the formulas in the theory are true given that interpretation. □

4.3 Logic Programming

Logic Programming (LP) is based on First-Order Logic and in particular on *clausal logic*, but has a slightly different semantics. Work on LP started in the 70's, in particular Kowalski in 1974 formalized the concept of logic programming language [18].

A **disjunctive logic program** is a set of clauses, also called **rules** of this form:

$$\mathtt{a}_1; \ldots; \mathtt{a}_n \ \mathtt{:-}\ \mathtt{b}_1, \ldots, \mathtt{b}_m. \tag{4.3}$$

where $n > 0$, $m \geq 0$, $\mathtt{a}_i$s are *atoms*, $\mathtt{b}_j$s are literals, the character ";" is equivalent to *disjunction* $\vee$, "," is equivalent to *conjunction* $\wedge$, and "." indicates the end of a clause.

If $n > 1$ the clause is also called **disjunctive clause**, whereas if $n = 1$ the clause is called **non-disjunctive** or **normal clause**. A **normal logic program** is a logic program composed only of normal clauses.

A **definite logic program** [19] is a logic program that has exactly one atom in the conclusion, i.e. $n = 1$, and all the $\mathtt{b}_j$s are positive literals, i.e. atoms. Formally, a definite logic program rule/clause is of the form:

$$\mathtt{a} \texttt{ :- } \mathtt{b}_1, \ldots, \mathtt{b}_m. \tag{4.4}$$

A two-valued **Herbrand interpretation** $\mathcal{I}$ of a logic program P is a subset of $hb(P)$. A Herbrand interpretation $\mathcal{I}$ represents a possible world where all the elements in $\mathcal{I}$ are true and the elements of $hb(P) \setminus \mathcal{I}$ are false. As in First-Order Logic, an Herbrand interpretation is a model if all the formulas in P evaluate to true in that interpretation.

In 1976, van Emden and Kowalski in [19] presented different semantics for definite logic programs. These are known as model-theoretic, procedural and fixpoint semantics. The *model-theoretic semantics* exploits the *Herbrand model intersection property* and defines the model of a logic program as the intersection of all Herbrand models of the logic program, i.e. the Least Herbrand Model (LHM). The LHM is equal to the least of all the Herbrand models w.r.t. set inclusion ordering, i.e. the model that makes the fewest atoms true. Intuitively, the LHM is the set of all ground atoms that are entailed by the definite logic program. The second one was named *procedural semantics* where it is possible to use a proof procedure called *linear resolution with selection function for definite logic programs* (*SLD-resolution*) that succeeds for the atoms true in the logic program. Finally, the *fixpoint semantics* is defined using the immediate consequence operator T_P, a mapping from Herbrand interpretations to Herbrand interpretations. All these semantics compute the same set of ground atoms that are logical consequences of the logic program.

LP semantics considers only the least model of an LP because it makes the **closed world assumption** [20] (CWA). Under the CWA, everything that is not inferred to be true is assumed to be false.

4.3.1 Prolog

Prolog stands for "PROgrammation en LOGique" (PROgramming in LOGic). It is the first logic programming language, developed in 1972 by Alain Colmerauer and Philippe Roussel at the University of Marseille by exploiting the ideas of Kowalski and van Emdem.

In this section we provide an informal description of SLD-resolution implemented in Prolog. For a formal treatment please refer to [18, 19].

First of all, we define some concepts. As mentioned before, a clause is **ground** if it does not contain variables. A *substitution* θ is an assignment of terms to variables $\theta = \{\mathtt{V}_1/\mathtt{t}_1, ..., \mathtt{V}_n/\mathtt{t}_n\}$, where $\mathtt{V}_i$ is a variable and $\mathtt{t}_i$ is the value associated with the variables. The application of a substitution to clause C (atom $\mathtt{a}$), is denoted with $C\theta$ ($\mathtt{a}\theta$). It means we are replacing the variables appearing in C ($\mathtt{a}$) with the corresponding values defined in θ. Given two atoms $\mathtt{a}$ and $\mathtt{b}$ and a substitution θ, we say that $\mathtt{a}$ and $\mathtt{b}$ can be *unified* if there exists a θ such that $\mathtt{a}\theta$ and $\mathtt{b}\theta$ are identical. The substitution

θ is the *most general unifier (mgu)* if there is no substitution ω that unifies `a` and (b) and such that $\theta = \omega\sigma$, where σ is a substitution.

SLD-resolution starts form a clause called *goal* or *query* that we want to resolve.

$$\mathtt{a}_1, \mathtt{a}_2, ..., \mathtt{a}_n$$

with $n > 0$.

Then, it iteratively selects a subgoal, i.e. an atom of the clause, and replaces this subgoal with the body of the clause contained in the program whose head can be *unified* with the selected subgoal. For example, if the selected subgoal is the first and the clause is

$$\mathtt{b}_0 \texttt{ :- } \mathtt{b}_1, ..., \mathtt{b}_m$$

where $m \geq 0$ and $\mathtt{b}_0$ can be unified with $\mathtt{a}_1$ through the mgu substitution θ, then the goal becomes

$$(\mathtt{b}_1, ..., \mathtt{b}_m, \mathtt{a}_2, ..., \mathtt{a}_n)\theta$$

Prolog's SLD-resolution differs from SLD-resolution because the subgoal selection strategy is fixed, the first subgoal on the left is always chosen. The execution ends when no more resolutions can be done, and in this case the query *fails*, or when the goal is empty, and in this case the query succeeds.

SLD-resolution was proven *sound*, i.e. the conclusions returned by the algorithm are logical consequences of the program. SLD-resolution itself is also *complete* for definite logic programs. If a query Q is a logical consequence of a program P, then there is a refutation of $P \cup \{Q\}$ by SLD-resolution. Conversely, Prolog's SLD-resolution is incomplete because the leftmost order in the choice of the next subgoal to prove can lead to infinite derivations.

In the following example, we graphically show the resolution of a query following Prolog's SLD-resolution.

Example 4.3.1
Consider the following Prolog program

```
uncle(X,Y) :- brother(X,Z),parent(Z,Y).   (4.5)
parent(X,Y) :- father(X,Y).               (4.6)
parent(X,Y) :- mother(X,Y).               (4.7)
mother(della,huey).                       (4.8)
brother(donald,della).                    (4.9)
```

Suppose the query is `uncle(donald,huey).`*, first the goal is rewritten using (4.5). Then,* `brother(donald,Z)` *unifies with clause (4.9) by using the substitution* $\theta_1 =$ {`Z/della`} *and is removed from the goal (the body of a fact is empty). At this point, the truth of* `parent(della,huey)` *must be proved. Here two possible ways can be tested, but only one results in the empty clause. All these steps are shown in Figure 4.1.*

In the next example we show the incompleteness of Prolog's SLD-resolution.

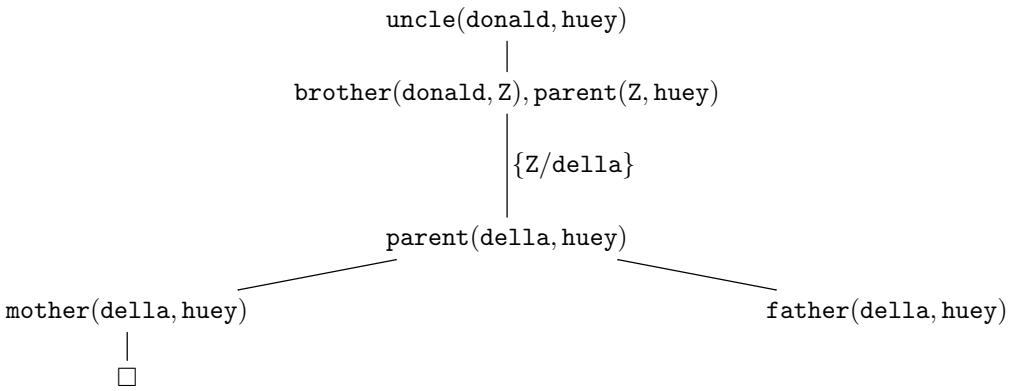


Figure 4.1: Prolog SLD resolution tree for the query `uncle(donald,huey)` w.r.t. the theory of Example 4.3.1.

Example 4.3.2
Consider the following program:

```
married(X,Y) :- married(Y,X).        (4.10)
married(a,b).                        (4.11)
```

with query `married(a,X)`*. The goal is unified with the head of rule (4.10) creating a new goal identical to the query. Then, an infinite number of resolution steps are performed. In this example, this issue can be avoided by moving rule (4.10) textually after fact (4.11), so that all refutations are found before going into an infinite cycle.*

4.3.2 Normal Logic Programs

In normal LPs we can have negative literals in the rule bodies that are usually interpreted using **negation as failure**[3], a non-monotonic inference rule that considers negation as failure to prove. The notation ***not*** `p`, may be read as "`p` is not provable", or "there is no proof for `p`". Note that ***not*** `p` is different from the classical, truth-functional logical negation ¬`p`.

Several alternative semantics for negation as failure exist. The most common are: **Clark's completion** [21], **stable models** [22] and the **well-founded semantics** [23, 24]. The first two semantics are two-valued, i.e. a literal can only be true or false, whereas the third one also allows for a third value ⊥ representing "don't know".

4.3.2.1 Clark's completion

Clark's completion [21] was the first attempt to deal with negation as failure. In this semantics, the original logic program in converted into a new program, called *completion* or *completed program*, and treated as a FOL theory. Then, only the literals, whether positive or negative, that are logical consequences of the completed program are considered true. In this semantics, negation as failure ***not*** in the original logic

[3]Also known as **default negation**.

program has the same meaning of the classical logic negation $\neg$ in the completed program.

To generate the completion of a logic program, in Clark's completion, we replace ***not*** with $\neg$, thereafter we collect all the rules having the same head predicate into a single rule whose body is a disjunction of conjunctions, then replace the symbol "`:-`" with "$\leftrightarrow$". Finally we add the Clark's equality theory, which are clauses for equality. Clark's completion can be used for *acyclic logic programs*.

Definition 4.9 Acyclic Program ([25])
A program P is **acyclic** if there is a function mapping $|\ | : hb(P) \rightarrow \mathbb{N}$, which maps each ground literal to a natural number, named *level*, so that for every rule r and every literal $\mathtt{l} \in body(r)$ we have that $|\mathtt{l}| < |head(r)|$, where $body(r)$ is the set of literals in the body of r and $head(r)$ is the atom head of r. □

For acyclic logic programs the completion has a single model (Theorem 2.5 [25]). For **cyclic logic program**, instead, the uniqueness of the model of the completion does not hold and the completed program can be inconsistent. SLDNF-resolution (SLD with Negation as Failure), which is the most common proof procedure for normal logic programs, is sound, but not complete in general, with respect to Clark's completion.

4.3.2.2 Stable Models

The stable model semantics, introduced by Gelfond and Lifschitz in [22], provides another semantics for normal logic programs and thus for negation as failure.

In this semantics, we first generate the Herbrand instantiation P_H of a logic program P, then a transformation on a given interpretation $\mathcal{I}$, called *stability transformation* and denoted as $\mathbf{S}(\mathcal{I})$, is performed. The stability transformation is divided into three phases:

1. For each rule instantiation in P_H, if it contains a *negative* subgoal ***not*** a such that $\mathsf{a} \in \mathcal{I}$, i.e. the negative subgoal is inconsistent with $\mathcal{I}$, then the rule instantiation is discarded.

2. Remove all negative subgoals from the rules of P', leaving a Horn program P''.

3. Obtain the least Herbrand model of P''.

A model $\mathcal{M}$ of a normal logic program P is **stable** if it is a fixed point of $\mathbf{S}$, that is, if $\mathcal{M} = \mathbf{S}(\mathcal{M})$. A stable model is also called **answer set**.

The stable model semantics is at the basis of the Answer Set Programming (ASP) research field. In ASP, this semantics has been extended for programs with arbitrary aggregates and complex reasoning systems have been proposed for this field [26].

4.3.2.3 Well-Founded Semantics

Before explaining the well-founded semantics [23, 24], we must provide some preliminary definitions.

Preliminary Definitions

Definition 4.10 Partial Order, Upper Bound, Lower Bound
A relation on a set S is a *partial order* if it is reflexive, antisymmetric and transitive. In the following, let S be a set with a partial order $\preceq$. $a \in S$ is an *upper bound* of a subset $X \subseteq S$ if $x \preceq a$ for all $x \in X$. Similarly, $b \in S$ is a *lower bound* of X if $b \preceq x$ for all $x \in X$. □

Definition 4.11 Least Upper Bound, Greatest Upper Bound
An element $a \in S$ is the *least upper bound* of a subset $X \subseteq S$ if a is an upper bound of X and, for all upper bounds a' of X, we have $a \preceq a'$. Similarly, $b \in S$ is the *greatest lower bound* of a subset $X \subseteq S$ if b is a lower bound of X and, for all lower bounds b' of X, we have $b' \preceq b$. The least upper bound of X is unique, if it exists, and is denoted by $lub(X)$. Similarly, the greatest lower bound of X is unique, if it exists, and is denoted by $glb(X)$. □

Definition 4.12 Complete Lattice
A partially ordered set L is a *complete lattice* if $lub(X)$ and $glb(X)$ exist for every subset $X \subseteq L$. We let $\top$ denote the top element $lub(L)$ and $\bot$ denote the bottom element $glb(L)$ of the complete lattice L. □

Definition 4.13 Monotonicity of a Mapping Operator, Least Fixed Point
Let L be a complete lattice and $\mathbf{T} : L \to L$ be a mapping. We say $\mathbf{T}$ is *monotonic* if $\mathbf{T}(x) \preceq \mathbf{T}(y)$, whenever $x \preceq y$. We say $a \in L$ is the *least fixed point* of $\mathbf{T}$ if a is a fixed point (that is, $\mathbf{T}(a) = a$) and for all fixed points b of $\mathbf{T}$ we have $a \preceq b$. Similarly, we define the *greatest fixed point*. □

Definition 4.14 Mapping Operator Iteration
Let L be a complete lattice and $\mathbf{T} : L \to L$ be monotonic. Then we define $\mathbf{T} \uparrow 0 = \bot$; $\mathbf{T} \uparrow \alpha = \mathbf{T}(\mathbf{T} \uparrow (\alpha - 1))$, if α is a successor ordinal; $\mathbf{T} \uparrow \alpha = lub(\{\mathbf{T} \uparrow \beta \mid \beta < \alpha\})$, if α is a limit ordinal; $\mathbf{T} \downarrow 0 = \top$; $\mathbf{T} \downarrow \alpha = \mathbf{T}(\mathbf{T} \downarrow (\alpha - 1))$, if α is a successor ordinal; $\mathbf{T} \downarrow \alpha = glb(\{\mathbf{T} \downarrow \beta \mid \beta < \alpha\})$, if α is a limit ordinal. □

Proposition 4.1
Let L be a complete lattice and $\mathbf{T} : L \to L$ be monotonic. Then $\mathbf{T}$ has a least fixed point, $lfp(\mathbf{T})$ and a greatest fixed point $gfp(\mathbf{T})$. □

Well-Founded Semantics
The well-founded semantics (WFS) [23, 24] assigns a three-valued model to a program, i.e. it identifies a consistent three-valued interpretation as the meaning of the program. A *three-valued interpretation* $\mathcal{I}$ of a logic program P is a pair $\langle \mathcal{I}_T, \mathcal{I}_F \rangle$, where $\mathcal{I}_T$ and $\mathcal{I}_F$ are subset of the Herbrand base $hb(P)$ and represent the set of true and false atoms respectively. A positive literal p is true in $\mathcal{I}$ if $\mathsf{p} \in \mathcal{I}_T$, and is false if $\mathsf{p} \in \mathcal{I}_F$. A negative literal ***not*** p is true in $\mathcal{I}$ if $\mathsf{p} \in \mathcal{I}_F$ and is false if $\mathsf{p} \in \mathcal{I}_T$. If $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_F \rangle$ is such that $\mathcal{I}_T \cap \mathcal{I}_F = \emptyset$ the we say that $\mathcal{I}$ is *consistent*, otherwise it is *inconsistent*. The union of two three-valued interpretations $\langle \mathcal{I}_T, \mathcal{I}_F \rangle \cup \langle \mathcal{J}_T, \mathcal{J}_F \rangle$ is defined as $\langle \mathcal{I}_T \cup \mathcal{J}_T, \mathcal{I}_F \cup \mathcal{J}_F \rangle$. Intersection is defined analogously.

The space of three-valued interpretations for a program P form a *complete lattice*, where the *partial order* relation $\preceq$ is defined as $\langle \mathcal{I}_T, \mathcal{I}_F \rangle \preceq \langle \mathcal{J}_T, \mathcal{J}_F \rangle$ if $\mathcal{I}_T \subseteq \mathcal{J}_T$ and $\mathcal{I}_F \subseteq \mathcal{J}_F$. Since the space of three-valued interpretations is a complete lattice there exist a least upper bound, $lub(X) = \bigcup_{\mathcal{I} \in X} \mathcal{I}$, and the greatest lower bound $glb(X) = \bigcap_{\mathcal{I} \in X} \mathcal{I}$.

We now define the two operators $\mathbf{T}_\mathcal{I}$ and $\mathbf{U}_\mathcal{I}$ as follows.

Definition 4.15
Let P be a normal logic program, $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_F \rangle$ be a three valued interpretation, where $\mathcal{I}_T$ and $\mathcal{I}_F$ are the sets of true and false atoms respectively, and let T and F be two sets of ground atoms. We define the operators $\mathbf{T}_\mathcal{I}(T) : \mathcal{P}(hb(P)) \rightarrow \mathcal{P}(hb(P))$ and $\mathbf{U}_\mathcal{I}(F) : \mathcal{P}(hb(P)) \rightarrow \mathcal{P}(hb(P))$ as

$\mathbf{T}_\mathcal{I}(T) = \{\mathtt{p} \mid \mathtt{p} \notin \mathcal{I}_T$; and there is a rule $\mathtt{q}$:- $\mathtt{l}_1, \ldots, \mathtt{l}_n$ in P, and a grounding substitution θ such that $\mathtt{p} = \mathtt{q}\theta$ and every $1 \leq i \leq n$, either $\mathtt{l}_i\theta$ is true in $\mathcal{I}$ or $l_i\theta \in T$ $\}$

$\mathbf{U}_\mathcal{I}(F) = \{\mathtt{p} \mid \mathtt{p} \notin \mathcal{I}_F$; and for every rule $\mathtt{q}$:- $\mathtt{l}_1, \ldots, \mathtt{l}_n$ in P, and a grounding substitution θ such that $\mathtt{p} = \mathtt{q}\theta$ there is some i, with $1 \leq i \leq n$, such that $\mathtt{l}_i\theta$ is false in $\mathcal{I}$ or $\mathtt{l}_i\theta \in F$ $\}$

□

We now define the operator $\mathbf{W}(\mathcal{I})$, which construct successive three-valued interpretations as follows.

Definition 4.16
Let P be a normal logic program, $\mathcal{I}$ be a three valued interpretation. We define the operator $\mathbf{W}(\mathcal{I}) : \mathcal{P}(hb(P)) \times \mathcal{P}(hb(P)) \rightarrow \mathcal{P}(hb(P)) \times \mathcal{P}(hb(P))$, where $\mathcal{P}$ denotes the power set operator, as

$$\mathbf{W}(\mathcal{I}) = \mathcal{I} \cup \langle lfp(\mathbf{T}_\mathcal{I}(\emptyset)), gfp(\mathbf{U}_\mathcal{I}(hb(P))) \rangle$$

□

$\mathbf{W}(\mathcal{I})$ is monotonic [23] and thus it has a least fixed point, which is indeed the well-founded model WFM of the logic program P

$$WFM(P) = lfp(\mathbf{W}(\mathcal{I}))$$

The well-founded model is the main semantics for normal logic programs. In the case of *definite logic programs*, the well-founded model is identical to the *Least Herbrand Model* (LHM). For these programs, the LHM is guaranteed to exist and be unique.

If a normal logic program is *acyclic* then the well-founded semantics, stable models and Clark's completion, coincide. (Theorem 1 in [27]) Moreover, if the program is *range restricted*[4] (see below), SLDNF resolution is a correct and complete procedure for answering queries in them [25].

[4]Range restriction avoids the foundering of the subgoals.

Definition 4.17 Range Restricted Program
A program is **range restricted** if for each rule all the variables appearing in the head of each rule also appear in positive literals in the body. □

The requirement of acyclicity is a quite strong restriction that rules out many interesting programs. However, Fages in [28] proved that if an logic program is *tight*, i.e. without positive cycles, then the Herbrand models of its Clark's completion [21] are minimal and coincide with the stable models of the original logic program. Any logic program can be transformed into a tight program that preserves the program's completion semantics. In [27] Riguzzi showed that, if a logic program P is *modular acyclic*, its well-founded model $WFM(P)$ is two-valued and coincides with the unique stable model and with the unique Herbrand model of Clark's completion of the program.

4.4 First-Order Logic vs Logic Programs

There is a crucial difference between LP and FOL since FOL does not make the CWA. For example the FOL theory $\{a \leftarrow b\}$, has 3 models $\{\neg a, \neg b\}$, $\{a, \neg b\}$ and $\{a, b\}$. The LP theory {`a :- b`} is a definite program, therefore it has only one well-founded model, that is its Least Herbrand Model $\emptyset$.

FOL and LP have different semantics and expressiveness. In fact, in FOL we can express that a given relation is transitive but we can't express a non-ground *transitive closures* (a.k.a. *inductive definitions*). For example, consider the following logic program written in Prolog (taken from [29]).

Example 4.4.1

```
edge(1,2).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

The least Herbrand model of this program is {`edge(1,2), path(1,2)`}, which corresponds to the transitive closure of the relation `edge/2`. The transitive closure of a relation[5] R is the minimal relation R^+ which contains R and is transitive. If we interpret the mentioned program in First-Order Logic we have a total of six possible Herbrand models:

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), \neg path(1,1), \neg path(2,1), \neg path(2,2)\}$

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), path(1,1), \neg path(2,1), \neg path(2,2)\}$

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), path(1,1), path(2,1), \neg path(2,2)\}$

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), \neg path(1,1), \neg path(2,1), path(2,2)\}$

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), path(1,1), \neg path(2,1), path(2,2)\}$

$\{\neg edge(1,1), edge(1,2), \neg edge(2,1), \neg edge(2,2), path(1,2), path(1,1), path(2,1), path(2,2)\}$

[5] A predicate is a relation between objects in the domain.

It can be seen that while `path` is transitive in each of these models, the transitive closure holds only in the first model.

4.5 Conclusions

In this chapter we illustrated the fundamentals of First-Order Logic (FOL) and Logic Programming (LP). In addition, we briefly described Prolog, the first and the most famous logic programming language, in the case of definite programs. For normal logic programs, i.e. logic programs with negations in the body, we have three different semantics: Clark's completion [21], stable models [22] and the well-founded semantics [23, 24]. Moreover we discussed the semantical differences between FOL and LP.

The limits of FOL expressiveness carry over to the probabilistic case: we can express transitive closure with Probabilistic Logic Programming languages like ProbLog [30] and LPADs [31] (see Chapter 6) but not with FOL-based languages like Markov Logic Networks [32]. This does not mean that formalisms based on First-Order Logic are not practical. Working with FOL-based formalisms is useful when much information is unknown and we want monotonic reasoning[6] or we want to obtain all the possible models of a FOL theory, for example in the Semantic Web, where the information could be distributed and not complete[7]. However, if we want transitive closures and CWA, logic programming could be the best choice.

In the next chapters we present the distribution semantics and two family of probabilistic logics: Probabilistic Logic Programming (PLP) languages (Chapter 6) and Probabilistic Description Logics (PDLs) (Chapter 9).

[6] Monotonic reasoning means that if we add a new axiom in the knowledge, the previously obtained logical consequences are still valid.

[7] Indeed we will see that OWL, used to represent knowledge in the Semantic Web, is a family of logical formalisms based on FOL.

Chapter 5

Distribution Semantics

In this chapter we present the **distribution semantics** [1].The aim of this part ot the thesis is to introduce probabilistic logical formalisms and this semantics underlies many Probabilistic Logic Programming languages (see Chapter 6) and it is the semantics on which DISPONTE is based. DISPONTE stands for "DIstribution Semantics for Probabilistic ONTologiEs" and assigns a meaning to Probabilistic Description Logics (see Chapter 9). After the introduction in Section 5.1 of the problems tackled in this chapter, we present the distribution semantics' foundations in Section 5.2. Section 5.3 concludes the chapter.

5.1 Introduction

In real world domains the information is often uncertain, hence it is of foremost importance to be able to model uncertainty and to reason over it. Moreover the diffusion of Logic Programming (LP) techniques made clear that an integration with probability theory was necessary. As a consequence, in the the last decades, several semantics were proposed that presented different probabilistic semantics for LP languages [33, 34, 35, 1]. Among them, two different approaches emerged, one that makes use of variants of the distribution semantics [1] and one that exploits Knowledge Base Model Construction (KBMC) [36, 37].

In the latter approach, the program is converted into a graphical model, usually a Bayesian network or a Markov network, in order to model probabilistic information and compute the probability of queries, whereas the former approach defines a probability distribution over normal logic programs, also called **worlds**. This distribution is then extended to a joint distribution over worlds and queries, from which the probability of a query, i.e. a ground fact, is computed by marginalization, i.e., by summing out the worlds.

Languages that apply a KBMC approach include Probabilistic Knowledge Bases [38], Bayesian Logic Programs [39], CLP(BN) [40] and the Prolog Factor Language [41]. These approaches specify a model through *features* that are associated with a real value, i.e. a probability value or weight.

The distribution semantics was first presented in 1995 by Taisuke Sato [1]. He presented a semantics applicable to *definite logic programs* and defined the basis for probabilistic inference and parameter learning. Later in [42, 43] the authors showed

that the distribution semantics works also with *normal logic programs* with function symbols. Nowadays the distribution semantics underlies many Probabilistic Logic Programming languages such as the Independent Choice Logic [44], PRISM [1], Logic Programs with Annotated Disjunctions [31] and ProbLog [45].

In the next section we briefly describe the distribution semantics, for further reading and deep understanding we refer to [1].

5.2 Formal Definition

Let F be a set of facts and R be a set of definite rules. $DB = F \cup R$ is a definite program, with countably many variables, function and predicate symbols, which respect the following conditions:

- DB is ground or it can be reduced to the set of all possible ground instantiations of the clauses.
- DB is countably infinite.
- DB satisfies the *disjoint condition* which imposes that no atom in F unifies with the head of a rule in R.

Let $A_1, A_2, \ldots$ be an arbitrary enumeration of ground atoms appearing in F. Each ground atom A_i is associated with a Boolean random variable which takes value 1 (if A_i is true) or 0 (if A_i is false). An interpretation ω for F is an assignment of truth to atoms A_i in F, it is identified as a possibly infinite vector $\omega = \langle x_1, x_2, \ldots \rangle$, where x_i is the truth value of A_i. Ω_F is the set of all interpretations for F and it is defined as a Cartesian product of $\{0, 1\}$s

$$\Omega_F = \prod_{i=1}^{\infty} \{0, 1\}_i \tag{5.1}$$

A *basic distribution* P_F for F is a probability measure on the algebra of the sample space Ω_F. The corresponding distribution function is $P_F^{(n)}(x_1, \ldots, x_n)$ for $n = 1, 2, \ldots$.

Each interpretation $\omega = \langle x_1, x_2, ... \rangle \in \Omega_F$ defines a set $F_\omega \subset F$ of true ground atoms, thus we can define a logic program $F_\omega \cup R$ and its least model $M_{DB}(\omega)$ which decides all truth values of atoms in DB.

Example 5.2.1
Given the finite program DB_1:

$$\begin{aligned} DB_1 &= F_1 \cup R_1 \\ F_1 &= \{A_1, A_2\} \\ R_1 &= \{B_1 \leftarrow A_1, B_1 \leftarrow A_2, B_2 \leftarrow A_2\} \end{aligned}$$

we have $\Omega_F = \{0, 1\} \times \{0, 1\}$ and $\omega = \langle x_1, x_2 \rangle \in \Omega_F$ means that A_i takes the truth value x_i $(i = 1, 2)$. M_{DB} is shown in Table 5.1.

Table 5.1: M_{DB_1} for the finite program DB_1.

$\omega = \langle x_1, x_2 \rangle$	$F_{1\omega}$	$M_{DB_1}(\omega)$
$\langle 0, 0 \rangle$	$\{\}$	$\{\}$
$\langle 1, 0 \rangle$	$\{A_1\}$	$\{A_1, B_1\}$
$\langle 0, 1 \rangle$	$\{A_2\}$	$\{A_2, B_1, B_2\}$
$\langle 1, 1 \rangle$	$\{A_1, A_2\}$	$\{A_1, A_2, B_1, B_2\}$

Table 5.2: P_{F_1} and P_{DB_1} for the finite program DB_1.

$\omega = \langle x_1, x_2 \rangle$	$P_{F_1}(\omega)$
$\langle 0, 0 \rangle$	0.2
$\langle 1, 0 \rangle$	0.3
$\langle 0, 1 \rangle$	0.4
$\langle 1, 1 \rangle$	0.1
others	0.0

$\omega = \langle x_1, x_2, y_1, y_2 \rangle$	$P_{DB_1}(\omega)$
$\langle 0, 0, 0, 0 \rangle$	0.2
$\langle 1, 0, 1, 0 \rangle$	0.3
$\langle 0, 1, 1, 1 \rangle$	0.4
$\langle 1, 1, 1, 1 \rangle$	0.1
others	0.0

Let $A_1, A_2, \dots$ be an enumeration of all atoms appearing in DB this time[1]. Ω_{DB}, similarly to Ω_F, represents the set of all possible interpretations for ground atoms appearing in DB and it is a Cartesian product of $\{0, 1\}$s. $\omega \in \Omega_{DB}$ determines the truth value of each ground atom.

Let us introduce the notation A_i^x for an atom A_i which means that $A_i^x = A_i$ if $x = 1$ and $A_i^x = \neg A_i$ if $x_i = 0$. We can now extend P_F to define the probability measure P_{DB} over Ω_{DB} as follows

$$[A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F = \{\omega \in \Omega_F | M_{DB}(\omega) \models A_1^{x1} \wedge \cdots \wedge A_n^{x_n}\} \tag{5.2}$$

$$P_{DB}^{(n)}(x_1, \dots, x_n) = P_F([A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F) \tag{5.3}$$

where $P_{DB}^{(n)}$ is the corresponding finite distribution function of P_{DB}. Intuitively, P_{DB} is identified with an possibly infinite joint distribution $P_{DB}(x_1, x_2, \dots)$ on the probabilistic ground atoms $A_1, A_2, \dots$ in the Herbrand base of DB. This way a program denotes a distribution in this semantics.

Example 5.2.2
Consider the program DB_1 in Example 5.2.1 and the distribution P_{F_1} shown in Table 5.2, $\omega = \langle x_1, x_2, y_1, y_2 \rangle \in \Omega_{DB_1}$ indicates that x_i is the value of A_i and y_j is the value of B_j, where $i, j = 1, 2$. P_{DB_1} can be computed from P_{F_1}. See Table 5.2.

Let G an arbitrary formula without free variables whose predicates are among DB, $[G]$ is defined as

$$[G] = \{\omega \in \Omega_{DB} \mid \omega \vDash G\}$$

[1]Note that this is is different from the previous definition. Previously we defined $A_1, A_2, \dots$ as an enumeration of all atoms in F.

$[G]$ contains all the possible worlds where G is satisfied. Then the probability of G is defined as $P_{DB}([G])$, which represents the probability mass assigned to the set of interpretations satisfying G.

5.3 Conclusions

The distribution semantics [1] was the main topic of this chapter. This semantics tries to syncretize logic with probability theory, in order to represent uncertain information.

This semantics underlies many Probabilistic Logic Programming languages, which are the topic of the next chapter.

Chapter 6

Probabilistic Logic Programming Languages

In this chapter, after a brief introduction (Section 6.1) we present two of the most famous Probabilistic Logic Programming languages based on distribution semantics: Logic Programs with Annotated Disjunctions (LPADs), illustrated in Section 6.2, and ProbLog presented in Section 6.3.

6.1 Introduction

The distribution semantics presented in Chapter 5 underlies many Probabilistic Logic Programming (PLP) languages such as Probabilistic Logic Programs [33], Probabilistic Horn Abduction [35], Independent Choice Logic [44], PRISM [1], pD [46], Logic Programs with Annotated Disjunctions (LPADs) [31], ProbLog [45, 30], P-log [47] and CP-logic [48].

The main difference between these logical languages is the definition of the distribution over logic programs. However, each language can be translated into the others using transformation algorithms which have linear complexity. Moreover, these languages are Turing complete, hence they are very expressive.

In this chapter we present Logic Programs with Annotated Disjunctions [31], because they have the most general syntax and are the knowledge representation language used in our PLP systems proposed in Chapter 12 and Chapter 15, and ProbLog [45, 30] for its simplicity.

6.2 Logic Programs with Annotated Disjunctions

This section presents the formalism of LPADs, introduced by J. Vennekens et al. in [31].

6.2.1 LPADs Syntax

An LPAD is a finite set of *annotated disjunctive clauses* of the form

$$\mathtt{h}_{i1} : \Pi_{i1}; \ldots ; \mathtt{h}_{in_i} : \Pi_{in_i} \ \mathtt{:-}\ \mathtt{b}_{i1}, \ldots, \mathtt{b}_{im_i}. \tag{6.1}$$

where i is the index of the rule, $\mathtt{b}_{i1}, \ldots, \mathtt{b}_{im_i}$ are literals, $\mathtt{h}_{i1}, \ldots \mathtt{h}_{in_i}$ are atoms and $\Pi_{i1}, \ldots, \Pi_{in_i}$ are annotations which are real numbers in the interval $[0, 1]$. This clause can be interpreted as "if $\mathtt{b}_{i1}, \ldots, \mathtt{b}_{im_i}$ is true, then $\mathtt{h}_{i1}$ is true with probability Π_{i1} or ... or $\mathtt{h}_{in_i}$ is true with probability Π_{in_i}." If $n_i = 1$ and $\Pi_{i1} = 1$ the clause is non-disjunctive, while if $n_i > 1$ then $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, there is an implicit atom *null* : $(1 - \sum_{k=1}^{n_i} \Pi_{ik})$ that does not appear in the body of any clauses of the program.

Example 6.2.1
The following LPAD T from [49] encodes a very simple model of the development of an epidemic or a pandemic:

C_1 = `epidemic : 0.6; pandemic : 0.3 :- flu(X),cold.`
C_2 = `cold : 0.7.`
`flu(david).`
`flu(robert).`

An epidemic or a pandemic may arise if somebody has the flu and the climate is cold. We are uncertain whether the climate is cold and we know for sure that David and Robert have the flu.

6.2.2 LPADs Semantics

For the sake of simplicity we consider only the case of LPADs without function symbols.

Definition 6.1 Atomic choice
An **atomic choice** is a selection of the k-th atom for a grounding $C_i\theta_j$ of a probabilistic clause C_i and is represented by the triple (C_i, θ_j, k), where θ_j is a substitution (a set of couples $\mathtt{V}_i/\mathtt{v}_i$, where $\mathtt{V}_i$ is variable and $\mathtt{v}_i$ is a constant) and $k \in \{1, \ldots, n_i\}$. An atomic choice represents an equation of the form $X_{ij} = k$ where X_{ij} is a random variable associated with $C_i\theta_j$. □

Definition 6.2 Consistency of an atomic choice
A set of atomic choices κ is *consistent* if $(C_i, \theta_j, k) \in \kappa, (C_i, \theta_j, m) \in \kappa$ implies $k = m$, i.e., only one head is selected for a ground clause. □

Definition 6.3 Composite choice
A **composite choice** κ is a consistent set of atomic choices. □

The probability of a composite choice κ is

$$P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik} \tag{6.2}$$

where Π_{ik} is the probability annotation of head k of clause C_i.

Definition 6.4 Selection
A **selection** σ is a total set of atomic choices (one atomic choice for every grounding of each probabilistic clause). □

A selection σ *identifies* a logic program w_σ called a *world*. The probability of w_σ is

$$P(w_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik} \tag{6.3}$$

Since the program does not contain function symbols, the set of worlds is finite $\mathcal{W} = \{w_1, \ldots, w_m\}$ and $P(w)$ is a distribution over worlds: $\sum_{w\in\mathcal{W}} P(w) = 1$.

We consider only sound LPADs where, for each selection σ, the well-founded model of the program w_σ is two-valued. We write $w_\sigma \models Q$ to mean that the query Q is true in the well-founded model of the program w_σ. Since the well-founded model of each world is two-valued, Q can only be true or false in w_σ.

We define the conditional probability of a query Q given a world as $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. It is now possible to define the probability of Q by using two rules of the theory of probability:

- *marginalization* or *sum rule*:

$$P(Q) = \sum_{w\in\mathcal{W}} P(Q, w)$$

- and *product rule*:

$$P(Q, w) = P(Q|w)P(w)$$

So the the probability of Q becomes:

$$P(Q) = \sum_{w\in\mathcal{W}} P(Q, w) = \sum_{w\in\mathcal{W}} P(Q|w)P(w) = \sum_{w\in\mathcal{W}:w\models Q} P(w) \tag{6.4}$$

Example 6.2.2
For the LPAD T of Example 6.2.1, clause C_1 has two groundings, $C_1\theta_1$ with $\theta_1 =$ {X/david} and $C_1\theta_2$ with $\theta_2 =$ {X/robert}, while clause C_2 has a single grounding $C_2\emptyset$. T has $3\times3\times2$ worlds, the query $Q =$ epidemic is true in 5 of them and its probability is $P(Q) = 0.6\cdot0.6\cdot0.7+0.6\cdot0.3\cdot0.7+0.6\cdot0.1\cdot0.7+0.3\cdot0.6\cdot0.7+0.1\cdot0.6\cdot0.7 = 0.588$.

It is often infeasible to find all the worlds where the query is true, so inference algorithms find, instead, *explanations* for the query, i.e. particular types of composite choices (read below).

A composite choice κ *identifies* a set of worlds $\omega_\kappa = \{w_\sigma|\sigma \in \mathcal{S}, \sigma \supseteq \kappa\}$, the set of worlds whose selection is a superset of κ, where $\mathcal{S}$ is the set of all the possible selections. The set of worlds *identified* by a set of composite choices K is defined as $\omega_K = \bigcup_{\kappa\in K} \omega_\kappa$.

Definition 6.5 Explanation (for PLP)
A composite choice κ is an **explanation** for a query Q if Q is entailed by every world of ω_κ. □

A set of composite choices K is *covering* Q if every world $w_\sigma \in \mathcal{W}$ in which Q is entailed is such that $w_\sigma \in \omega_K$, i.e. $\omega_K = \{w_\sigma|\sigma \in \mathcal{S} \wedge w_\sigma \models Q\}$. In other words a covering set K identifies all the worlds in which Q succeeds. *The set of all the explanations for Q is a covering set of Q.*

Example 6.2.3
Consider the LPADs in Example 6.2.1. A set of composite choices K that covers the query `epidemic` *is*

$$K = \{\kappa_1, \kappa_2\}$$
$$\kappa_1 = \{(C_1, \{\texttt{X/david}\}, 1), (C_2, \emptyset, 1)\}$$
$$\kappa_2 = \{(C_1, \{\texttt{X/robert}\}, 1), (C_2, \emptyset, 1)\}$$

Two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent. For Example $\kappa_1 = \{(C_i, \theta_j, 1)\}$ and $\kappa_2 = \{(C_i, \theta_j, 0)\}$ are incompatible. A set K of composite choices is *pairwise incompatible* if for all $\kappa_1 \in K$, $\kappa_2 \in K$, $\kappa_1 \neq \kappa_2$ implies κ_1 and κ_2 are incompatible.

The *probability of a pairwise incompatible set of composite choices* K is defined as follows:

$$P(K) = \sum_{\kappa \in K} P(\kappa) \tag{6.5}$$

Two set of composite choices K_1 and K_2 are *equivalent* if they identify the same set of worlds, i.e., if $\omega_{K_1} = \omega_{K_2}$. Given a query Q and its covering set of composite choices K, then K identifies a set of worlds $\omega_K = \{w_\sigma | \sigma \in \mathcal{S} \wedge w_\sigma \models Q\}$. Then we have that

$$P(Q) = \sum_{w_\sigma \in \omega_K} P(w_\sigma) = P(\omega_K) = P(K) \tag{6.6}$$

If K is pairwise incompatible

$$P(Q) = \sum_{w_\sigma \in \omega_K} P(w_\sigma) = P(\omega_K) = P(K) = \sum_{\kappa \in K} P(\kappa) \tag{6.7}$$

Example 6.2.4
Consider the LPAD of Example 6.2.1. In Example 6.2.3 we found a covering set of explanations for the query $Q =$ `epidemic.`*, but those explanation are not pairwise incompatible therefore we cannot compute the probability of the query with those explanations by using Equation (6.7). In fact*

$$P(\kappa_1) + P(\kappa_2) = 0.6 \cdot 0.7 + 0.6 \cdot 0.7 = 0.84 \neq 0.588 = P(Q)$$

where $P(Q)$ was computed in Example 6.2.2.

Suppose now that we have the following covering set of explanations K' for the query $Q =$ `epidemic.`

$$K' = \{\kappa'_1, \kappa'_2\}$$
$$\kappa'_1 = \{(C_1, \{\texttt{X/david}\}, 1), (C_1, \{\texttt{X/robert}\}, 0), (C_2, \emptyset, 1)\}$$
$$\kappa'_2 = \{(C_1, \{\texttt{X/robert}\}, 1), (C_2, \emptyset, 1)\}$$

The explanations are pairwise incompatible, then we can use Equation (6.7) to compute the probability of K', which is equal to $P(Q)$. In fact

$$P(K') = 0.6 \cdot 0.4 \cdot 0.7 + 0.6 \cdot 0.7 = 0.588 = P(Q)$$

To compute the conditional probability $P(Q|E)$ of a query Q given evidence E, we can use the definition of conditional probability, $P(Q|E) = P(Q, E)/P(E)$, and compute first the probability of Q, E (the sum of probabilities of worlds where both Q and E are true) and the probability of E and then divide the two.

If an LPAD contains function symbols, a more complex definition of the semantics is necessary: since the number of groundings is infinite, a world would be obtained by making an infinite number of choices and so its probability, the product of infinite numbers all smaller than one and bounded away from one, would be 0. In this case we have to work with sets of worlds and use Kolmogorov's definition of probability space, see [50].

6.3 ProbLog

ProbLog [45, 30] assigns probabilities to facts, so it is possible to define a *joint distribution over facts*, which, according to [1], we are able to extend to a *joint distribution over the set of possible logic programs.*

6.3.1 ProbLog Syntax

A ProbLog program $\mathcal{T}$ is composed of a normal logic program $\mathcal{C}$ and a set of probabilistic facts $\mathcal{F}$. Each probabilistic fact is of the form

$$p_i :: F_i$$

where $F_i \in \mathcal{F}$ is an atom and p_i is a probability, i.e. $p_i \in [0, 1]$. This means that every grounding $F_i\theta_j$ of F_i is a Boolean random variable that assumes true value with probability p_i and false with probability $1 - p_i$. The set of all the groundings of the probabilistic facts in $\mathcal{F}$ is denoted as $\mathcal{F}^G$.

A world w obtained from a ProbLog program is the union of the normal logic program $\mathcal{C}$ and a subset $\mathcal{F}_w^G$ that contains a selection of ground probabilistic facts chosen within the set of all the groundings of the probabilistic facts $\mathcal{F}^G$ ($\mathcal{F}_w^G \subset \mathcal{F}^G$).

The probability of a world is computed by multiplying p_i for each probabilistic fact F_i included in the world and $1 - p_j$ for each probabilistic fact F_j not included in the world. The probability of a query Q is computed by marginalization as for LPADs.

Example 6.3.1
Let us consider the ProbLog program corresponding to the LPAD of Example 6.2.1.

```
epidemic :- flu(X), epid(X), cold.
pandemic :- flu(X),\+ epid(X), pand(X), cold.
flu(david).
flu(robert).
```

$F_1 =$ `0.7 :: cold.`

$F_2 =$ `0.6 :: epid(X).`

$F_3 =$ `0.3 :: pand(X).`

where `\+` *is the negation as failure* ***not****. This program models the fact that if somebody has the flu and the weather is cold there is the possibility that an epidemic or a pandemic arises. We are uncertain about whether the climate is cold, but we know for sure that David and Robert have the flu. The facts* `epid(X)` *and* `pand(X)` *can be considered as "probabilistic activators" of the effects in the head given that the causes (*`flu(X)` *and* `cold`*) are present.*

Fact F_1 has only one grounding, while facts F_2 and F_3 have two groundings obtained by assigning to `X` *the value* `david` *or* `robert`*. From F_2 we obtain* `epid(david)` *and* `epid(robert)`*, while from F_3 we obtain* `pand(david)` *and* `pand(robert)`*. $\mathcal{T}$ has 5 different ground probabilistic facts and thus 32 worlds. The query* `epidemic` *is true in 12 of them and its probability is $P($`epidemic`$) = 0.588$. For the sake of brevity, we do not report here the formula with the probability of all the worlds where the query is true, but we show two examples of possible worlds. One world where the query is true is (note that we show only the probabilistic facts):*

$$\{\texttt{cold,epid(david), epid(robert), pand(david), pand(robert)}\}$$

whose probability is $0.7 \cdot 0.6 \cdot 0.6 \cdot 0.3 \cdot 0.3 = 0.02268$.

Another different world in which the query is true is:

$$\{\texttt{cold,epid(david),pand(robert)}\}$$

whose probability is $0.7 \cdot 0.6 \cdot 0.3 = 0.126$.

6.4 Conclusions

In this chapter we illustrated two of the most famous Probabilistic Logic Programming languages based on distribution semantics: LPADs (Section 6.2) and ProbLog (Section 6.3). In particular, LPADs are th probabilistic logical formalism used by the inference system `cplint`, discussed in Chapter 12, and by the distributed structure learning algorithm SEMPRE, presented in Chapter 17.

The languages following the distribution semantics differ in the way they define the distribution over logic programs. However, each language can be translated into the others using transformation algorithms which have linear complexity [51].

The next chapter introduces description logics, a different family of logical formalism for knowledge representation, and OWL, a logical language for the Semantic Web based on description logics.

Chapter 7

Description Logics and OWL

In this chapter we discuss Description Logics (DLs) and the Web Ontology Language (OWL). DLs are a family of knowledge representation formalisms and OWL is a language for the Semantic Web that provide several concrete syntaxes for Description Logics.

The chapter is organized as follows. Sections 7.1 provides an introduction. Section 7.2 illustrate the abstract syntax, the naming scheme and the semantics of description logics. Section 7.5 shows the relationship between description logics and First-Order Logic. Section 7.6 provide a brief overview of OWL and the Semantic Web. Finally Section 7.7 concludes the chapter.

7.1 Introduction

Description Logics (DLs) are a family of **knowledge representation** (KR) formalisms based on KL-ONE [52]. They are drawing an increasing interest thanks to their use in the **Semantic Web**. Therefore extending DLs with probability could be very useful to represent uncertain information in the Semantic Web.

During the 1970s several approaches to knowledge representation were proposed and they are sometimes divided in two main categories: logic-based and non-logic-based formalisms. The former evolved out of the intuition that predicate calculus could be used unambiguously to capture facts about the world [52], they were more formal and hence more general-purpose. The latter, like *frames* and *semantic network*, were often developed by building on cognitive notions [52] and therefore they were more human-understandable, but they usually lacked a formal logic-based semantics[1]. To overcome this deficiency, some knowledge representation systems were proposed, these systems were initially called *terminological systems*, then *concept languages* and finally **description logics**. The main reason for using DLs rather than predicate logic is that DLs are carefully tailored such that they combine interesting language constructs with decidability of the reasoning problems [53]. In effect, FOL is undecidable, whereas description logics are usually decidable fragments of FOL. However, they use a different

[1]Indeed in the non-logical approaches, knowledge is represented by means of some *ad hoc* data structures, and reasoning is accomplished by similarly *ad hoc* procedures that manipulate the structures

terminology from FOL. They use the terms *concepts*, *roles* and *individuals* for *unary predicates*, *binary predicates* and *constants*.

In this chapter we provide a brief overview of description logics, the theory behind them. Moreover we illustrate the Web Ontology Language (OWL), a family of languages that implements various description logics, i.e. OWL provides concrete syntaxes for DLs. For more details about description logics we refer to [52, 53, 54]. Instead for further information about Semantic Web and OWL syntax we refer to the W3C World Wide Web Consortium online site [55, 56].

7.2 Description Logics

An **ontology** is a formal and explicit description of a domain of interest. The use of ontologies solves term ambiguity and clarifies domain peculiarities, e.g. the word *leg* could mean a part of human body (if we are in a medical context) or it could mean a part of a table (if we are in the context of carpentry).

An ontology describes the *concepts* of the domain of interest and their relations with a formalism such that it is possible for a computing machine to use that ontology through specific programs, called *reasoners*.

Descriptions logics provide a logical formalism for knowledge representation. In DLs information is stored in *knowledge bases* (KB). They are usually divided into two parts: the *intensional knowledge*, or *ontology*, and the *extensional knowledge*. The former introduces the *terminology* by relating concepts and roles and it provides a vocabulary for the domain of interest. The latter contains *assertions* or specific information regarding concept and role membership of individuals.

DLs are useful in all the domains where it is necessary to represent information and to perform inference on it, such as software engineering, medical diagnosis, digital libraries, databases and Web based informative systems. They possess nice computational properties such as decidability and (for some DLs) low complexity.

7.3 Syntax

The basic syntactic building blocks are the following fiur disjoint sets:

- **individuals**, that correspond to all names used to denote individual entities (be they persons, objects or anything else) in the domain, like mary, boston, italy; they are equivalent to FOL constants;

- **atomic concepts**, which denote types, categories, or classes of entities, usually characterized by common properties, e.g., Cat, Country, Doctor; they are equivalent to FOL unary predicates;

- **atomic roles** a.k.a. **abstract roles**, which denote binary relationships between individuals of a domain, e.g., hasParent, loves, locatedIn; they are equivalent to FOL binary predicates;

- **datatype roles**, which denote binary relationship between individuals and data values such as strings and numbers, i.e. assign data values to individuals, e.g. hasAge, hasName; they are equivalent to FOL binary predicates.

In the following sections, we will use C and D to denote arbitrary concepts, R and S to denote arbitrary roles, A to denote an atomic concept, and n to denote a non-negative integer.

7.3.1 Concept and Role Constructors

Each DL language has its own expressiveness determined by which constructors and axioms are allowed.

7.3.2 Concept Constructors

For concepts, the most used constructors are **union** ($\sqcup$), **intersection** ($\sqcap$) and **negation** ($\neg$). For negation, we must distinguish between atomic concept negation and complex concept negation. A **complex concept** is an atomic concept or a concept defined by a set of concepts (atomic or not) combined by constructors. In addition, many DLs define two specific concepts, the **universal concept** *top* ($\top$), which is equivalent to $A \sqcup \neg A$, and the **inconsistent concept** *bottom* ($\bot$), which represent the empty concept and it is equivalent to $A \sqcap \neg A$. $\top$ represent the set of all the individuals, whereas $\bot$ is the empty set to which no individuals belong.

Other common constructors are the *quantification constructors*. These constructors can be classified into *unqualified* and *qualified*. We can have **existential role restrictions** ($\exists R$ and $\exists R.C$ unqualified and qualified respectively, where C is a (complex) concept) and **universal role restrictions** ($\forall R$ and $\forall R.C$). The qualified existential restriction $\exists R.C$ indicates all those individuals that have at least a relation R with an individual belonging to C. In the above expression, the individuals belonging to C are called *role fillers*. The qualified universal role restriction $\forall R.C$, instead, indicates all those individuals that, if they have a relation R, it is only with individuals belonging to C. The unqualified constructors can be seen as a particular case of qualified constructors with the fillers belong to $\top$, i.e. $\exists R$ and $\forall R$ are equivalent to $\exists R.\top$ and $\forall R.\top$ respectively. Some DLs also support *cardinality restriction*, that can be qualified or unqualified, that place cardinality restrictions on the roles relating instances of a concept to instances of some other concept. Cardinality restrictions bound the number of individuals

In some DL languages there is also the possibility to define concepts by enumeration of individuals

Let $\mathbf{I}$, $\mathbf{A}$, $\mathbf{R_A}$, $\mathbf{R_D}$ be the sets of *individuals*, *atomic concepts*, *abstract roles* and *datatype roles* respectively. All these sets are pairwise disjoint. Moreover, let C and D be concepts and $R \in \mathbf{R_A}$ then the following are concepts as well:

- $\top$, top concept, contains everything;
- $\bot$, bottom concept, contains nothing;
- $A \in \mathbf{A}$;

- for every finite set $\{a_1, ..., a_n\} \in \mathbf{I}$ of individuals names, $\{a_1, ..., a_n\}$ is a concept called *nominal*;
- $C \sqcap D$, the intersection of two concepts;
- $C \sqcup B$, the union of two concepts;
- $\neg C$, the negation of a concept;
- $\exists R$ and $\exists R.C$, the unqualified and qualified existential restriction on a role;
- $\forall R$ and $\forall R.C$, the unqualified and qualified universal restriction on a role;
- $\geq nR$, $\leq nR$ and $= nR$ for an integer $n \geq 0$, unqualified number restriction on a role;
- $\geq nR.C$, $\leq nR.C$ and $= nR.C$ for an integer $n \geq 0$, qualified number restriction on a role.

If P is an n-ary datatype predicate and $T \in \mathbf{R_D}$, then:

- $\exists T.P$, the datatype existential restriction on a role;
- $\forall T.P$, the datatype universal restriction on a role.

7.3.3 Role constructors

Role constructors take role and/or concept descriptions and transform them into more complex role description. Let $\mathbf{R}$ be the set of all roles and R and S be two roles, then the following are roles

- U, universal role;
- $R \in \mathbf{R}$;
- $R \sqcap S$, the intersection of two roles;
- $R \sqcup S$, the union of two roles;
- $\neg R$, the negation of a role;
- $R \circ S$, the composition of two roles, used to define chain of roles;
- R^-, the inverse of role R;
- R^+, the transitive closure of role R;
- R^*, the reflexive-transitive closure of role R;
- $R|_C$, the role restriction of R, which defines a subrole of R whose range is restricted to the individuals belonging to the concept C.

7.3.4 Knowledge Base

A KB based on a Description Logic contains two kinds of information, *intensional knowledge* and *extensional knowledge*.

The former contains the *Terminological Box* (TBox) and the *Role Box* (RBox) and models general information about the domain, normally contains immutable information and statements which describe the main properties of concepts and relationships.

The latter, composed by the *Assertional Box* (ABox), contains information that is specific to the problem, that may change over time and that is related to the individuals of the domain.

7.3.4.1 TBox

The TBox contains axioms related to concepts. Let C and D be concepts. A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion* and *concept equivalence* axioms.

Concept inclusion axioms a.k.a. *concept subsumption axioms*, introduce a hierarchy among concepts. These axioms specify a *is-a* relationship between two different concepts. For example, we can state that a man is a person as

$$\mathsf{Man} \sqsubseteq \mathsf{Person}$$

the axiom above must be read as: "Person subsumes Man" or equivalently "Man is subsumed by Person"

Concept equivalence axioms assert equality between two concepts. Definitions are often used to associate a symbolic name to complex concepts. In these cases a single definition for a symbolic name is admitted in the TBox. For example, we can define the concepts Parent as the union between the concepts Mother and Father as

$$\mathsf{Parent} \equiv \mathsf{Mother} \sqcup \mathsf{Father}$$

Concept equivalence axioms can be expressed with subsumptions as $C \equiv D$ is equivalent to $C \sqsubseteq D$ and $D \sqsubseteq C$.

7.3.4.2 RBox

The RBox is a set of axioms concerning the roles contained in the KB. We use $\mathbf{R_A}^-$ to denote the set of all inverses of roles in $\mathbf{R_A}$.

Role inclusion axioms are of the form $R \sqsubseteq S$, where $R, S \in \mathbf{R_A} \cup \mathbf{R_A}^-$ or $R, S \in \mathbf{R_D}$.

Role equivalence axioms are of the form $R \equiv S$, which is an abbreviation for $R \sqsubseteq S$ and $S \sqsubseteq R$.

Role chain axioms are of the form $R_1 \circ R_2 \sqsubseteq R_3$, where $R_1, R_2, R_3 \in \mathbf{R_A} \cup \mathbf{R_A}^-$. For example, to model the fact the father of the father of an individual is a grandparent the following axiom can be used:

$$\mathsf{fatherOf} \circ \mathsf{fatherOf} \sqsubseteq \mathsf{grandParentOf}$$

Transitivity axioms are of the form $Trans(R)$, where $R \in \mathbf{R_A}$ or $R \in \mathbf{R_D}$. They mean that if x is related to y and y is related to z with role R, then x is R-related to z. For example, if the role brotherOf is transitive and the axioms brotherOf(luca, andrea) and brotherOf(andrea,giovanni) are given, then we can conclude that brotherOf(luca,giovanni) is also true. They are equivalent to $R \circ R \sqsubseteq R$

Functional axioms are of the form $Funct(R)$. They mean that, for each object x, there can be only one object y in relation with x through R. There cannot be two distinct y_1 and y_2 such that we have $R(x, y_1)$ and $R(x, y_2)$. For example, consider the relation childOfFather, and consider the kid luca. If we have childOfFather(luca, f_1) and childOfFather(luca, f_2), then we can conclude:

1. f_1 and f_2 are the same person, i.e. the father of *luca*
2. if $f_1 \neq f_2$ is also stated, then the KB is inconsistent[2]

An *RBox* $\mathcal{R}$ consists of a finite set of *role inclusion axioms*, *roles chain axioms*, plus axioms that define the characteristics of roles such as *transitivity axioms* and *functional axioms*. Which axioms may be present in an RBox depends on the expressive power of the Description Logic, in some cases the KB does not contain the RBox, in $\mathcal{ALC}$ KBs.

7.3.4.3 ABox

An *ABox* (Assertional Box) contains information about the individuals of the problem domain. It defines which classes each individual belongs to and how the individuals are related to each other.

Let a, b be individuals and v be a data value, an *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms*, *role membership axioms*, *datatype role membership axioms*, *equality axioms* and *inequality axioms*.

Concept membership axioms are of the form $a : C$, where C is a concept. They state that a belongs to the concept C.

Role membership axioms are of the form $(a, b) : R$, where $R \in \mathbf{R_A}$. They state that b is R-related to or is a filler of the role R for a.

Datatype role membership axioms are of the form $(a, v) : T$, where $T \in \mathbf{R_D}$. They state that v is T-related to a.

Equality axioms are of the form $a = b$. They state that a and b define the same individual.

Inequality axioms are of the form $a \neq b$. They state that a and b are different individuals. This axiom is extremely important in order to make the Unique Name Assumption (see Definition 7.6).

[2]We are considering a biological father.

7.3.5 Nomenclature

As mentioned before, DLs are a family of FOL-based KR languages. Many varieties have evolved during the years, which differ in terms of expressive power and syntactic structures. There is a well-established naming scheme that associates particular syntactic constructors to letter for composing the name of the DL. The naming schema can be summarized as follows:

$$((\mathcal{AL}\ [\mathcal{C}] \mid \mathcal{FL} \mid \mathcal{EL} \mid \mathcal{S})\ [\mathcal{H}] \mid \mathcal{SR})\ [\mathcal{O}]\ [\mathcal{I}]\ [\mathcal{F} \mid \mathcal{E} \mid \mathcal{U} \mid \mathcal{N} \mid \mathcal{Q}]\ [^{+} \mid {}^{*}]\ [(\mathbf{D})]$$

In this scheme, the round brackets form a group (except for (**D**)), the square brackets indicate optional symbols that cannot appear on their own and the '|' represents alternatives. It is worth noting that the naming schema below it is not an *official* standard, but a standard *de facto*.

The symbols used in the nomenclature of DLs are defined as follows

- $\mathcal{AL}$ is the abbreviation of *attributive language*. It is often considered as the base language and allows atomic negation ($\neg A$), union ($\sqcup$) and intersection ($\sqcap$) as well as universal ($\forall R.C$) and unqualified existential ($\exists R.\top$) quantification.
- $\mathcal{ALC}$ is the abbreviation of *attributive language with complements*. $\mathcal{C}$ extends $\mathcal{AL}$ by allowing negation of complex concept ($\neg C$).
- $\mathcal{FL}$ is the contraction of *frame based description language*. It allows concept intersections, universal restrictions, unqualified existential quantifications and role restrictions. $\mathcal{FL}$ has two sublanguages: $\mathcal{FL}^{-}$, obtained by disallowing the use of role restrictions, and $\mathcal{FL}_0$, that is a sublanguage of $\mathcal{FL}^{-}$ obtained by forbidding unqualified existential quantifications, the bottom ($\bot$) and top ($\top$) concepts. $\mathcal{FL}^{-}$ is equivalent to $\mathcal{AL}$ without atomic negation.
- $\mathcal{EL}$ allows the use of existential quantifiers, concept intersections and the $\top$ (top) concept. It disallows unions, complements, universal quantifiers and axioms regarding roles such as role subsumptions. $\mathcal{EL}^{+}$ is an extension which allows the use of role inclusion axioms. $\mathcal{EL}^{++}$ is an alias for $\mathcal{ELRO}$.
- $\mathcal{S}$ extends the logic $\mathcal{ALC}$ by allowing the definition of transitive roles.
- $\mathcal{H}$ extends $\mathcal{ALC}$ and $\mathcal{S}$ by role hierarchies, thus it allows role inclusion axioms.
- $\mathcal{SR}$ extends $\mathcal{S}$ by allowing the definition of complex role inclusions, i.e. hierarchies between complex roles, e.g. $R_1 \circ R_2 \sqsubseteq S$ means that $R_1 \circ R_2$ is subrole of S.
- $\mathcal{O}$ allows the use of enumerations in the definition of concepts, i.e. the use of nominals in the definition of concepts, for example the definition of MontyPython can be {graham, john, terry, eric, terry, michael}.
- $\mathcal{I}$ enables the definition of inverse roles, e.g. R^{-} is the inverse role of R: $(a, b) : R$ iff $(b, a) : R^{-}$.
- $\mathcal{F}$ allows to define that a role R is functional, i.e. has at most one filler, which is equivalent to the axiom $\top \sqsubseteq 1R$.

- $\mathcal{E}$ means that the DL features qualified existential role restrictions.
- $\mathcal{U}$ allows union between concepts.
- $\mathcal{N}$ means that the definition of unqualified number role restrictions is allowed, i.e., $\leq nR$, $\geq nR$ and $= nR$.
- $\mathcal{Q}$ means that qualified number role restrictions can be defined, i.e., $\leq nR.C$, $\geq nR.C$ and $= nR.C$.
- $^+$ allows to define transitive closures of roles, e.g. R^+ is the transitive closure of R.
- * allows to define reflexive-transitive closures of roles, e.g. R^* is the transitive closure of R
- (**D**) allows datatype properties, such as numbers of strings.

Table 7.1 shows some of the cited DL constructors, for each constructor is reported its DL language that allows to express it. Transitive and reflexive–transitive closure are the only constructors that cannot be expressed in FOL.

Table 7.1: Some DL constructors with their associated DL language symbols.

Constructor	**Syntax**	**Languages**
Intersection	$C \sqcap D$	$\mathcal{EL} \mid \mathcal{AL} \mid \mathcal{FL}_0$
Qualified universal role restriction	$\forall R.C$	$\mathcal{AL} \mid \mathcal{FL}_0$
Top	$\top$	$\mathcal{EL} \mid \mathcal{AL} \mid \mathcal{FL}^-$
Bottom	$\bot$	$\mathcal{AL} \mid \mathcal{FL}^-$
Unqualified existential role restriction	$\exists R$ (i.e. $\exists R.\top$)	$\mathcal{AL} \mid \mathcal{FL}^-$
Role restriction	$R \mid_C$	$\mathcal{FL}$
Qualified existential role restriction	$\exists R.C$	$\mathcal{EL} \mid \mathcal{AL} \mid \mathcal{E}$
Union	$C \sqcup D$	$\mathcal{AL} \mid \mathcal{U}$
Atomic negation	$\neg A$	$\mathcal{AL}$
Negation	$\neg C$	$\mathcal{C}$
Unqualified number restriction	$\geq nR$ $\leq nR$ $= nR$	$\mathcal{N}$
Qualified number restriction	$\geq nR.C$ $\leq nR.C$ $= nR.C$	$\mathcal{Q}$
Inverse role	R^-	$\mathcal{I}$
Nominal	$\{x, y, z\}$	$\mathcal{O}$
Transitive closure	R^+	$_+$
Reflexive-transitive closure	R^*	$_*$

7.4 Semantics

Usually the semantics of a DL knowledge base $\mathcal{K}$ is assigned in a set-theoretic way, where every concept is interpreted as a set of individuals and every role as a set of pairs of individuals. We give now some definitions regarding the semantics.

Definition 7.1 Interpretation $\mathcal{I}$ for DLs without datatypes (D)
An interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of

- a non-empty set $\Delta^{\mathcal{I}}$, called the **domain** of $\mathcal{I}$, which contains all the individuals of the domain;
- an **interpretation function** $\cdot^{\mathcal{I}}$, that assigns an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to each $a \in \mathbf{I}$, a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ to each $C \in \mathbf{A}$ and a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each $R \in \mathbf{R}$, where $\mathbf{I}$, $\mathbf{A}$ and $\mathbf{R}$ are respectively the set of individuals, atomic concepts and atomic roles.

□

If the DL allows the use of datatypes, then the definition of interpretation given above must be extended to take into account also a **datatype theory** which is associated to $\mathcal{I}$. First of all we provide a definition of datatype theory.

Definition 7.2 Datatype theory of a DL
A datatype theory $\mathcal{D} = (\Delta^{\mathcal{D}}, \cdot^{\mathcal{D}})$ is defined by

- a non-empty datatype domain $\Delta^{\mathcal{D}}$,
- a mapping function $\cdot^{\mathcal{D}}$ which assigns to each data value an element of $\Delta^{\mathcal{D}}$, to each elementary datatype a subset of $\Delta^{\mathcal{D}}$, and to each datatype predicate[3] of arity n a relation over $\Delta^{\mathcal{D}}$ of arity n.

□

Definition 7.3 Interpretation $\mathcal{I}$ for DLs with datatypes (D)
Let $\mathbf{I}$, $\mathbf{A}$, $\mathbf{R_A}$ and $\mathbf{R_D}$ be respectively the set of individuals, atomic concepts, abstract roles and datatype roles, which are pairwise disjoint. An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ relative to a datatype theory $\mathcal{D} = (\Delta^{\mathcal{D}}, \cdot^{\mathcal{D}})$ is composed of a non-empty domain $\Delta^{\mathcal{I}}$ that is disjoint from $\Delta^{\mathcal{D}}$, and an interpretation function $\cdot^{\mathcal{I}}$ which maps each $a \in \mathbf{I}$ to an element of $\Delta^{\mathcal{I}}$, each $C \in \mathbf{A}$ to a subset of $\Delta^{\mathcal{I}}$, each $R \in \mathbf{R_A}$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, each $T \in \mathbf{R_D}$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$, and every data value, datatype, datatype predicate to the same value assigned by $\cdot^{\mathcal{D}}$. □

The mapping $\cdot^{\mathcal{I}}$ for DL constructors is reported in Tables 7.2-7.4, where $R^{\mathcal{I}}(x) = \{y \mid (x,y) \in R^{\mathcal{I}}\}$, $R^{\mathcal{I}}(x, C) = \{y \mid (x,y) \in R^{\mathcal{I}}, y \in C^{\mathcal{I}}\})$, $\#X$ denotes the cardinality of the set X, $T^{\mathcal{I}}(x) = \{y \mid y \in \Delta^{\mathcal{D}}, (x,y) \in T^{\mathcal{I}}\}$, $(R^{\mathcal{I}})^0 = \{(x,x) \mid x \in \Delta^{\mathcal{I}}\}$ and $(R^{\mathcal{I}})^{n+1} = (R^{\mathcal{I}})^n \circ R^{\mathcal{I}}$. Table 7.2 reports the semantics of the most common concept and individual constructors. Table 7.3 shows the semantics of the most common role constructors. Table 7.4, instead, illustrates the semantics of the most common datatype and data value constructors.

The satisfaction of an axiom E in an interpretation $\mathcal{I}$, denoted by $\mathcal{I} \models E$, is defined as follows

[3] In [52] they are called *concrete predicates*.

Table 7.2: Syntax and semantics of common concept and individual constructors.

Constructor	**Syntax**	**Semantics**
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Intersection	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Qualified universal role restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(x) \subseteq C^{\mathcal{I}}\}$
Qualified existential role restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(x) \cap C^{\mathcal{I}} \neq \emptyset\}$
Unqualified number restriction	$\geq nR$	$(\geq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x) \geq n\}$
	$\leq nR$	$(\leq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x) \leq n\}$
	$= nR$	$(= nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x) = n\}$
Qualified number restriction	$\geq nR.C$	$(\geq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x, C) \geq n\}$
	$\leq nR.C$	$(\leq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x, C) \leq n\}$
	$= nR.C$	$(= nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#R^{\mathcal{I}}(x, C) = n\}$
Nominal	$\{a, b, c\}$	$\{a, b, c\}^{\mathcal{I}} = \{a^{\mathcal{I}}, b^{\mathcal{I}}, c^{\mathcal{I}}\}$

Table 7.3: Syntax and semantics of common role constructors.

Constructor	**Syntax**	**Semantics**
Universal role	U	$U^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Atomic role/abstract role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Intersection	$R \sqcap S$	$(R \sqcap S)^{\mathcal{I}} = R^{\mathcal{I}} \cap S^{\mathcal{I}}$
Union	$R \sqcup S$	$(R \sqcup S)^{\mathcal{I}} = R^{\mathcal{I}} \cup S^{\mathcal{I}}$
Negation	$\neg R$	$(\neg R)^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \setminus R^{\mathcal{I}}$
Inverse role	R^-	$(R^-)^{\mathcal{I}} = \{(y, x) \mid (x, y) \in R^{\mathcal{I}}\}$
Composition	$R \circ S$	$(R \circ S)^{\mathcal{I}} = \{(x, y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists z.(x, z) \in R^{\mathcal{I}} \wedge (z, y) \in S^{\mathcal{I}}\}$
Role restriction	$R\|_C$	$(R\|_C)^{\mathcal{I}} = \{(x, y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(x) \subseteq C^{\mathcal{I}}\}$
Transitive closure	R^+	$(R^+)^{\mathcal{I}} = \bigcup_{n \geq 1} (R^{\mathcal{I}})^n$
Reflexive-transitive closure	R^*	$(R^*)^{\mathcal{I}} = \bigcup_{n \geq 0} (R^{\mathcal{I}})^n$

Table 7.4: Syntax and semantics of common datatype and data value constructors

Constructor	**Syntax**	**Semantics**
Datatype	D	$D^{\mathcal{D}} = D^{\mathcal{I}} \subseteq \Delta^{\mathcal{D}}$
Datatype role	T	$T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$
Negation	$\neg D$	$(\neg D)^{\mathcal{D}} = (\neg D)^{\mathcal{I}} = \Delta^{\mathcal{D}} \setminus D^{\mathcal{D}}$
Data value	v	$v^{\mathcal{I}} = v^{\mathcal{D}}$
Data enumeration	$\{u, w, v\}$	$\{u, w, v\}^{\mathcal{I}} = \{u^{\mathcal{I}}, w^{\mathcal{I}}, v^{\mathcal{I}}\}$
Qualified universal datatype role restriction	$\forall T.D$	$(\forall T.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid T^{\mathcal{I}}(x) \subseteq D^{\mathcal{D}}\}$
Qualified existential datatype role restriction	$\exists T.D$	$(\exists T.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid T^{\mathcal{I}}(x) \cap D^{\mathcal{D}} \neq \emptyset\}$

Definition 7.4 Axiom satisfaction

- A concept inclusion axiom $\mathcal{I} \models C \sqsubseteq D$ is satisfied by $\mathcal{I}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.
- A concept assertion axiom $\mathcal{I} \models a : C$ is satisfied by $\mathcal{I}$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$.
- A role assertion axiom $\mathcal{I} \models (a, b) : R$ is satisfied by $\mathcal{I}$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.
- An equality axiom $\mathcal{I} \models a = b$ is satisfied by $\mathcal{I}$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$.
- A inequality axiom $\mathcal{I} \models a \neq b$ is satisfied by $\mathcal{I}$ iff $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.
- A transitivity axiom $\mathcal{I} \models Trans(R)$ is satisfied by $\mathcal{I}$ iff $R^{\mathcal{I}}$ is transitive.
- A role inclusion axiom $\mathcal{I} \models R \sqsubseteq S$ is satisfied by $\mathcal{I}$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$.
- A datatype role assertion axiom $\mathcal{I} \models (a, v) : T$ for a data value v is satisfied by $\mathcal{I}$ iff $(a^{\mathcal{I}}, v^{\mathcal{D}}) \in T^{\mathcal{I}}$.

□

An interpretation $\mathcal{I}$ is a *model* of an axiom E if $\mathcal{I}$ satisfies E, i.e. if E is true with respect to $\mathcal{I}$. $\mathcal{I}$ *satisfies* a set of axioms $\mathcal{E}$, denoted by $\mathcal{I} \models \mathcal{E}$, iff $\mathcal{I} \models E$ for all $E \in \mathcal{E}$.

Definition 7.5 Model of a DL KB $\mathcal{K}$
An interpretation $\mathcal{I}$ *satisfies* a knowledge base $\mathcal{K}$, denoted $\mathcal{I} \models \mathcal{K}$, iff $\mathcal{I}$ satisfies all the boxes contained in $\mathcal{K}$, i.e. if $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, then $\mathcal{I} \models \mathcal{K}$ iff $\mathcal{I}$ satisfies $\mathcal{T}$ and $\mathcal{A}$, while if $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$, then $\mathcal{I}$ have also to satisfy $\mathcal{R}$. In this case we say that $\mathcal{I}$ is a *model* of $\mathcal{K}$. □

A knowledge base $\mathcal{K}$ is *satisfiable* iff there exists an interpretation $\mathcal{I}$ that satisfies $\mathcal{K}$. An axiom E is *entailed* by $\mathcal{K}$, denoted $\mathcal{K} \models E$, iff every interpretation that satisfies $\mathcal{K}$ also satisfies E.

We provide now the definition of the Unique Name Assumption.

Definition 7.6 Unique Name Assumption (UNA)
The *Unique Name Assumption* states that individuals with different names have to be interpreted as different individuals, i.e., that $a \neq b$ with $a, b \in N_I$ implies $a^{\mathcal{I}} \neq b^{\mathcal{I}}$. □

In more recent DLs, the UNA is not made but can be explicitly stated by adding $a \neq b$ to the ABox for every couple of distinct individuals in the KB. OWL provides the construct AllDifferent to ease the definition of different individuals.

An example of DL KB is shown in Example 7.4.1.

Example 7.4.1
The following is inspired by [57]. For simplicity we make the UNA.

$$\mathsf{Swallow} \sqsubseteq \mathsf{Bird} \tag{7.1}$$
$$\mathsf{MigrantBird} \sqsubseteq \mathsf{Bird} \tag{7.2}$$
$$\mathsf{NonMigrantBird} \sqsubseteq \mathsf{Bird} \tag{7.3}$$
$$\mathsf{EuropeanSwallow} \sqsubseteq \mathsf{Swallow} \sqcap \mathsf{MigrantBird} \tag{7.4}$$
$$\mathsf{AfricanSwallow} \sqsubseteq \mathsf{Swallow} \sqcap \mathsf{NonMigrantBird} \tag{7.5}$$
$$\exists \mathsf{trasported.Thing} \sqsubseteq \mathsf{Coconut} \tag{7.6}$$
$$\top \sqsubseteq \forall \mathsf{trasported.Bird} \tag{7.7}$$
$$\mathsf{TransportableCoconut} \equiv \exists \mathsf{transported.AfricanSwallow} \sqcup \geq 2\,\mathsf{transported.EuropeanSwallow} \tag{7.8}$$
$$\mathsf{TransportableCoconut} \sqsubseteq \mathsf{Coconut} \tag{7.9}$$
$$\mathsf{CoconutInEurope} \sqsubseteq \mathsf{Coconut} \tag{7.10}$$
$$\mathsf{CoconutInEurope} \equiv \geq 2\,\mathsf{transported.EuropeanSwallow} \tag{7.11}$$
$$\mathsf{fedor} : \mathsf{Coconut} \tag{7.12}$$
$$\mathsf{transported}(\mathsf{fedor}, \mathsf{ivan}) \tag{7.13}$$
$$\mathsf{transported}(\mathsf{fedor}, \mathsf{aleksej}) \tag{7.14}$$
$$\mathsf{dmitrij} : \mathsf{AfricanSwallow} \tag{7.15}$$
$$\mathsf{ivan} : \mathsf{EuropeanSwallow} \tag{7.16}$$
$$\mathsf{aleksej} : \mathsf{EuropeanSwallow} \tag{7.17}$$

The TBox is composed by terminological axioms 7.1-7.11. The subclass axioms 7.1-7.5 state that swallows are birds, that we can have migrant and non-migrant birds and that European swallows are migrant birds, whereas the African ones are non-migrant. The two terminological axioms 7.6-7.7 define that the role transported *has* Coconut *as domain (7.6) and* Bird *as range (7.7). The axioms affirm that a coconut is transportable if is carried by at least an African swallow or at least two European swallows and that if a coconut is transported by at least two European swallows, then the coconut belongs to the class of coconuts in Europe.*

The ABox is composed by assertional axioms 7.12-7.17, which state that Fedor is a coconut, Dmitrij is an African swallow, Ivan and Aleksej, instead, are European swallows and that Fedor The Coconut was transported by Ivan and Aleksej.

The fact that Fedor is a coconut in Europe is not explicit, but can be inferred by means of a reasoner (see Chapter 8).

7.4.1 Decidability of Description Logics

Each DL is *decidable* if the problem of checking the satisfiability of a KB is decidable.

Allowing arbitrary roles in cardinality restriction concepts is known to lead to undecidability [58].

In particular, a DL is decidable iff there are no cardinality restrictions on transitive roles and on roles that have transitive subroles [59, 60].

Role chains introduce some issues too:

- Arbitrary role chain axioms lead to undecidability. For ensuring decidability the following restrictions must be imposed:
  - there must be a strict linear order $\prec$ on roles
  - the set of role chain axioms must be *regular*, i.e., the set has to contain only role chain axioms of the following forms:

$$R \circ R \sqsubseteq R$$
$$S^- \sqsubseteq R$$
$$S_1 \circ S_2 \circ ... \circ S_n \sqsubseteq R$$
$$R \circ S_1 \circ S_2 \circ ... \circ S_n \sqsubseteq R$$
$$S_1 \circ S_2 \circ ... \circ S_n \circ R \sqsubseteq R$$

    where $S_i \prec R$ for all $i = 1, 2, \ldots, n$.

- In $\mathcal{SHIQ}$ (and $\mathcal{SHOIQ}$), the combination of role chain axioms with cardinality constraints may lead to undecidability. For ensuring decidability, qualified number restrictions has to be restricted to certain roles that were called *simple roles* [59]. In this context, a role is called *simple* if it is neither transitive nor has transitive sub-roles.

  $\mathcal{SROIQ}(\mathbf{D})$ was introduced by Horrocks et al. in [61] and it is of particular importance because it is semantically equivalent to OWL 2 DL (see Section 7.6). This DL allows to define qualified cardinality role restrictions, transitive roles and complex role inclusion axioms. In the context of $\mathcal{SROIQ}(\mathbf{D})$, the definition of *simple role* has to be slightly modified, and simple roles must appear not only in qualified number restrictions, but in several other constructs as well. Intuitively, non-simple roles are those that are implied, directly or indirectly, by a role chain.

  Definition 7.7 Simple role in $\mathcal{SROIQ}(\mathbf{D})$ [61]
  Given a role R, its *simplicity* is inductively defined as follows:

  - R is simple if it does not occur on the right hand side of a role inclusion axiom in $\mathcal{R}$, i.e. there is no role chain axiom of the form: $S_1 \circ S_2 \circ \cdots \circ S_n \sqsubseteq R$;
  - an inverse role R^- is simple if R is, and
  - if R occurs on the right hand side of a role inclusion axiom in $\mathcal{R}$, then R is simple if, for each $S \sqsubseteq R$, S is a simple role.

  □

Example 7.4.2
Consider the following KB $\mathcal{K}$:

$$\begin{array}{ll} Q \circ P \sqsubseteq R & R \circ P \sqsubseteq R \\ R \sqsubseteq S & P \sqsubseteq R \\ Q \sqsubseteq S & \end{array}$$

P and Q are simple, whereas R and S are non-simple.

It is well known that the complexity of $\mathcal{SROIQ}$ is N2ExpTime [62].

7.5 Description Logics and First-Order Logic

In this section we discuss the relationship between DLs and First-Order Logic (FOL). In particular we discuss the relationship of $\mathcal{SROIQ}$ with FOL. For a more detailed analysis see [63] and Chapter 4 of [52].

Many DLs can be seen as fragments of First-Order Logic (possibly with equality or counting quantifiers) and therefore DLs like $\mathcal{SROIQ}$ are less expressive than FOL [63]. However, the expressiveness of DLs that include the transitive closure goes beyond first-order logic, indeed transitive closure cannot be expressed in FOL due to the Compactness Theorem. When representing knowledge bases, the main reason for using DLs rather than FOL is that most DLs are actually *decidable*.

Following the definitions provided by Borgida in [63]

Definition 7.8 Semantic equivalence of a transformation
A concept C and its translation $\pi(C)(x)$ are said to be *equivalent* if and only if, for all interpretations[4] $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ and all $a \in \Delta^{\mathcal{I}}$, we have

$$a \in C^{\mathcal{I}} \Leftrightarrow \mathcal{I} \models \pi(C)(a) \tag{7.18}$$

□

Definition 7.9 Expressiveness of a language $\mathcal{L}$
A language $\mathcal{L}_2$ is *as expressive as* language $\mathcal{L}_1$, if there is a translation π from $\mathcal{L}_1$ to $\mathcal{L}_2$ such that for every sentence L in $\mathcal{L}_1$, $\pi(L)$ is *equivalent* to L. Two languages are *equally expressive* if each is as expressive as the other. □

Let $\mathcal{L}^k$ be the set of FOL formulas that have only unary and binary predicates and that can be expressed using at most k variables[5]. We can extend the languages $\mathcal{L}^k$ with *counting quantifiers* $\exists^{\leq n}$, $\exists^{\geq n}$, for every positive integer n, obtaining the languages $\mathcal{C}^k$, i.e. FOL over unary and binary predicates with counting quantifiers. We define $\mathcal{L}^k_{CNT}$ as a restricted subset of $\mathcal{C}^k$ where in any subformula $\exists^{\leq n}.\psi$ or $\exists^{\geq n}.\psi$, ψ has no more than two free variables.

[4] We view interpretations both as DL and FOL interpretations.
[5] $\mathcal{L}^k$ is a fragment of FOL.

Theorem 7.1 [63]
$\mathcal{L}^3_{CNT}$ is equally expressive as any DL without the transitive closure.

$\mathcal{SROIQ}$ does not allow to express transitive closures and therefore $\mathcal{SROIQ}$ KBs can be translated into FOL theories.

Following we provide a translation of $\mathcal{SROIQ}$ DL into FOL (to be precise, into $\mathcal{L}^3_{CNT}$) extending the translations provided in [64, 52]. The translation of concepts is given by two mapping functions π_x and π_y. The translation is recursively defined as follows

$$\begin{aligned}
\pi_x(A) &= A(x)\\
\pi_x(\neg C) &= \neg\pi_x(C)\\
\pi_x(C \sqcap D) &= \pi_x(C) \wedge \pi_x(D)\\
\pi_x(C \sqcup D) &= \pi_x(C) \vee \pi_x(D)\\
\pi_x(\exists R.C) &= \exists y.R(x,y) \wedge \pi_y(C)\\
\pi_x(\exists R^-.C) &= \exists y.R(y,x) \wedge \pi_y(C)\\
\pi_x(\forall R.C) &= \forall y.R(x,y) \rightarrow \pi_y(C)\\
\pi_x(\forall R^-.C) &= \forall y.R(y,x) \rightarrow \pi_y(C)\\
\pi_x(\{a\}) &= (x = a)\\
\pi_x(\geq nR.C) &= \exists^{\geq n} y.R(x,y) \wedge \pi_y(C)\\
\pi_x(\geq nR^-.C) &= \exists^{\geq n} y.R(y,x) \wedge \pi_y(C)\\
\pi_x(\leq nR.C) &= \exists^{\leq n} y.R(x,y) \wedge \pi_y(C)\\
\pi_x(\leq nR^-.C) &= \exists^{\leq n} y.R(y,x) \wedge \pi_y(C)\\
\pi_x(= nR.C) &= \exists^{= n} y.R(x,y) \wedge \pi_y(C)\\
\pi_x(= nR^-.C) &= \exists^{= n} y.R(y,x) \wedge \pi_y(C)
\end{aligned}$$

where

$$\exists^{\geq n} y.R(y,x) = \exists y_1,\ldots,y_n.\bigwedge_{i\neq j} y_i \neq y_j \wedge \bigwedge_i R(x,y_i)$$

$$\exists^{\leq n} y.R(x,y) = \forall y_1,\ldots,y_{n+1}.\bigwedge_{i\neq j} y_i \neq y_j \rightarrow \bigvee_i \neg R(x,y_i)$$

and $\exists^{=n} y.R(x,y)$ is defined as a conjunction the previous two π_y is obtained from π_x by replacing x with y and vice-versa.

The translation of the most common DL axioms is shown in Table 7.5.

Table 7.5: Correspondence between DL axioms and their translation into FOL. Functions π_x and π_y are exploited to translate the concepts contained in the axioms.

Axiom	**Translation**
$C \sqsubseteq D$	$\forall x.\pi_x(C) \rightarrow \pi_x(D)$
$a : C$	$C(a)$
$(a, b) : R$	$R(a, b)$
$a = b$	$a = b$
$a \neq b$	$a \neq b$
$R \sqsubseteq S$	$\forall x, y.R(x, y) \rightarrow S(x, y)$
$R_1 \circ ... \circ R_n \sqsubseteq S$	$\forall x_i, 0 \leq i \leq m.R_1(x_0, x_1) \wedge ... \wedge R_n(x_{m-1}, x_m) \rightarrow S(x_0, x_m)$
$Trans(R)$	$\forall x, y, z.R(x, z) \wedge R(z, y) \rightarrow R(x, y)$

7.6 The OWL Ontology Language

The **Semantic Web** is an evolution of World Wide Web (some people talk about Web 3.0). It encourages the inclusion of information and data, called *semantic content*, inside web pages and other published documents, using a suitable format so that it can be extracted and used in automatic reasoning. The W3C provides this definition of Semantic Web [65]:

> *The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.*

Its final purpose is to permit a more coherent and organized usage of the available information. It allows to build more advanced search engines that base their search also on semantics rather than only on syntax.

The *Web Ontology Language* (OWL) is a family of knowledge representation languages for authoring ontologies or knowledge bases. The OWL family contains many species, serializations, syntaxes and specifications with similar names. The most important specifications are OWL and OWL 2[6]

The OWL Web Ontology Language was published in 2004 and now is known as OWL 1 [66]. OWL 2 [55] is an extension and revision of OWL 1. The OWL 2 specification is managed by the World Wide Web Consortium (W3C) and since December 2012 it became a W3C recommendation. It keeps a full backward compatibility with OWL 1.

The development of this language is motivated by the Semantic Web activity, indeed OWL is part of the *Semantic Web Stack* (Figure 7.1) that illustrates the architecture of the Semantic Web [66]:

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.

[6]There also exists an intermediate specification called OWL 1.1

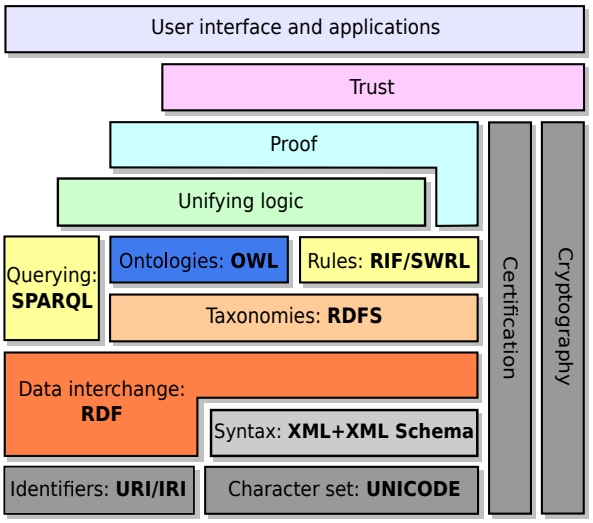


Figure 7.1: The Semantic Web Stack.

- XML Schema is a language for providing and restricting the structure and content of elements within XML documents and also extends XML with datatypes.
- RDF is a data model for objects ("resources") and relations between them and provides a simple semantics for this data model. It can be represented with an XML syntax.
- RDF Schema extends RDF and is a vocabulary for describing properties and classes of RDF-based resources, with semantics for hierarchies of such properties and classes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes, cardinality, equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.
- SPARQL is a protocol and query language for querying semantic web data sources.
- RIF is the W3C Rule Interchange Format. It's an XML language for representing Web rules which computers can execute.

OWL allows the user to write ontologies that describe the knowledge of a domain of interest by means of classes, roles and individuals. Such formalized knowledge can be automatically handled by a computer by means of an automatic reasoner.

OWL 2 DL is semantically equivalent to $\mathcal{SROIQ}(\mathbf{D})$ [54]. OWL 2 significantly extends the set of built-in datatypes of OWL 1 [67]: OWL 2 now supports `owl:boolean`, `owl:string`, `xsd:integer`, `xsd:dateTime`, `xsd:hexBinary`, and a number of datatypes derived from these by placing various restrictions on them. OWL 2 also provides a *datatype restriction* construct, which allows new datatypes to be defined by restricting the built-in datatypes in various ways. For example, the following expression defines a

Table 7.6: Terminology comparison of FOL, DL, and OWL.

FOL	DL	OWL
Unary predicate	Atomic concept	Class name
Formula with one free variable	Concept	Class
Binary predicate	Atomic role	Property name
Formula with two free variables	Role	Property
Constant	Individual	Individual
Sentence	Axiom	Axiom
Signature	Vocabulary or signature	Vocabulary or signature
Theory	Knowledge base	Ontology

new datatype by specifying a lower bound of 18 on the datatype `xsd:integer`

```
DatatypeRestriction(xsd:integer xsd:minInclusive 18)
```

The datatype restriction construct can be seen as a unary datatype predicate and it seems that OWL 2 does not support n-ary datatype predicates with $n > 1$ [68].

FOL, DLs, and OWL are strictly related, but each of them uses a different terminology. DL concepts correspond to FOL unary predicates and OWL classes, DL roles to FOL binary predicates and OWL properties, DL individuals to FOL constants and OWL individuals. Table 7.6 shows the terminology comparison of FOL, DL, and OWL. It is worth noting that in DL terminology the term ontology indicates the intensional knowledge, i.e. the union of TBox and RBox, whereas in OWL terminology it indicates the whole knowledge base.

7.6.1 OWL Syntax

There are many different types of syntaxes of OWL [55]

- *RDF/XML* syntax [69], as the name suggests, is based on XML. It allows writing down an RDF graph, and thus an OWL ontology. This syntax is the only one that a tool is obligated to support in order to be OWL 2 compliant.

- *Turtle* syntax [70] allows writing down an RDF graph, and thus an OWL ontology, in a compact textual form.

- *Manchester* syntax [71, 72] is an OWL syntax that is designed to be human readable and easily understandable even for non-logicians.

- *Functional-Style* syntax [73] is designed for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners.

- *OWL XML* syntax [74] is another syntax based on XML. An OWL ontology written with this syntax can be easily processed by a machine due to XML, but it is not very human readable.

Table 7.7 shows an example of an axiom expressed in these syntaxes.

Table 7.7: DL Axiom Woman $\sqsubseteq$ Person in different OWL 2 syntaxes.

Syntax	**Axiom**
DL syntax	Woman $\sqsubseteq$ Person

RDF/XML syntax	`<owl:Class rdf:about="Woman">`
	`  <rdfs:subClassOf rdf:resource="Person"/>`
	`</owl:Class>`

Manchester syntax	`Woman SubClassOf:  Person`
Turtle syntax	`:Woman rdfs:subClassOf :Person .`
Functional-Style syntax	`SubClassOf( :Woman :Person )`

OWL XML syntax	`<SubClassOf>`
	`  <Class IRI="Woman"/>`
	`  <Class IRI="Person"/>`
	`</SubClassOf>`

The set of the most common expressions supported by OWL, in DL and Manchester OWL syntax, is summarized in Table 7.8.

Table 7.9 shows some OWL axioms and how they can be mapped to DL and Manchester OWL syntax.

7.6.2 OWL sublanguages

The first version of OWL defined three different sublanguages of increasing complexity and expressiveness:

OWL Lite is based on $\mathcal{SHIF}(\mathbf{D})$ DL and supports classification hierarchies and simple constraints. Superclasses in concept inclusion axioms cannot be arbitrary class expressions but only named classes, i.e. simple concepts. Moreover, it admits cardinality restrictions with cardinality values of 0 or 1 only. Its main goal is to represent thesauri and taxonomies.

OWL DL extends OWL Lite. It allows all the constructors and the axioms permitted by the $\mathcal{SHOIN}(\mathbf{D})$ DL language. OWL DL is a language meant for users that want the maximum expressiveness of OWL 1 while maintaining computational completeness and decidability.

OWL Full has a highly expressive semantics that extends OWL DL. OWL Full contains all the OWL language constructs and provides free, unconstrained use of RDF constructs. For example, classes can be seen as both collections of individuals and single individuals. OWL Full is not decidable and the presence of tools able to support complete reasoning is implausible.

Table 7.8: Most common OWL expressions in DL and Manchester OWL syntax.

OWL expression	**DL syntax**	**Manchester syntax**
`Thing`	$\top$	`Thing`
`Nothing`	$\bot$	`Nothing`
`intersectionOf`	$C_1 \sqcap \cdots \sqcap C_n$	C_1 `and ... and` C_n
`unionOf`	$C_1 \sqcup \cdots \sqcup C_n$	C_1 `or ... or` C_n
`complementOf`	$\neg C$	`not` C
`oneOf`	$\{x1, \ldots, x_n\}$	{x_1, ..., x_n}
`allValuesFrom`	$\forall R.C$	R `only` C
`someValuesFrom`	$\exists R.C$	R `some` C
`hasValue`	$\exists R.\{x\}$	R `value` {x}
`minCardinality`	$(\geq nR)$	R `min` n
`maxCardinality`	$(\leq nR)$	R `max` n
`inverse`	R^-	`inverse` R

Table 7.9: Most common OWL axioms in DL and Manchester OWL syntax.

Axiom	**DL syntax**	**Manchester syntax**
`subClassOf`	$C \sqsubseteq D$	C `SubClassOf:` D
`equivalentClass`	$C \equiv D$	C `EquivalentTo:` D
`disjointWith`	$C \sqsubseteq \neg D$	C `DisjointWith:` D
`sameAs`	$\{x\} \equiv \{y\}$	x `SameAs:` y
`differentFrom`	$\{x\} \sqsubseteq \neg\{y\}$	x `DifferentFrom:` y
`subPropertyOf`	$R \sqsubseteq S$	R `SubPropertyOf:` S
`equivalentProperty`	$R \equiv S$	R `EquivalentTo:` S
`domain`	$\forall R.\top \sqsubseteq C$	R `domain` C
`range`	$\top \sqsubseteq \forall R.C$	R `range` C
`inverseOf`	$S \equiv R^-$	S `inverseOf:` R
`TransitiveProperty`	$R \circ R \sqsubseteq R$	R `Characteristics: Transitive`
`FunctionalProperty`	$\top \sqsubseteq (\leq 1R)$	R `Characteristics: Functional`
`SimmetricProperty`	$R \equiv R^-$	R `Characteristics: Symmetric`

Figure 7.2 show the organization of the three OWL 1 sublanguages.

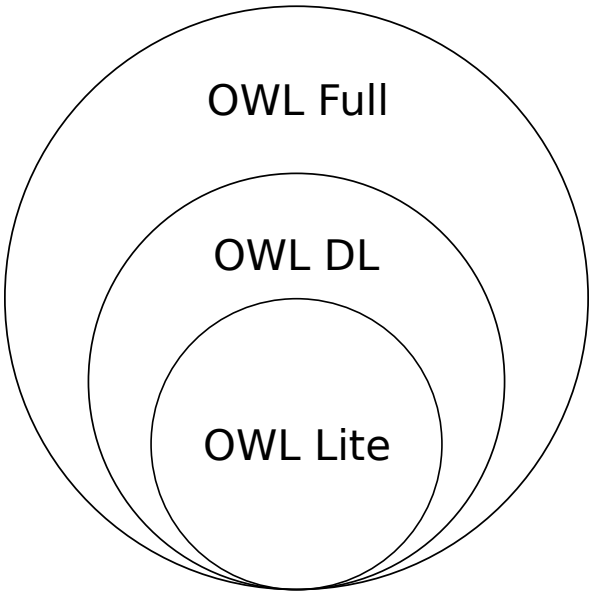


Figure 7.2: OWL 1 sublanguages.

In 2008 the W3C OWL Working Group published the specifications of the successor of OWL, called OWL 2. OWL 2 is also equipped with five different sublanguages:

OWL 2 EL corresponds to $\mathcal{EL}^{++}$. In this language, reasoning can be performed in a time that is polynomial in the size of the ontology. OWL 2 EL was expressly defined for applications with a very large number of classes and/or properties. This sublanguage was defined in order to represent large medical and biochemical ontologies, such as Gene Ontology[7] or SNOMED-CT[8] where there are thousands of classes.

OWL 2 QL is a sublanguage the offers a simplified support to queries on large amounts of instance data. It allows to keep data in relational databases and reasoning can be performed by means of query languages. It allows to express role inclusion axioms and inverse properties but disallows, for instance, the use of universal quantifiers.

OWL 2 RL allows to handle rules, such as *if-then-else* constructs. It makes use of standard rule languages. In this way queries can be answered by means of rule-based reasoning engines.

OWL 2 DL based on $\mathcal{SROIQ}(\mathbf{D})$ [54], includes all the three previous sublanguages. It is more expressive OWL DL by allowing qualified cardinal restrictions and complex role inclusion axioms, on the other hand reasoning is more complex.

[7] `http://geneontology.org/`
[8] `http://www.ihtsdo.org/snomed-ct`

OWL 2 Full like OWL Full, contains all the OWL 2 language constructs and provides free, unconstrained use of RDF constructs. It includes OWL 2 DL. Like its previous version, it is not decidable and the support to complete reasoning is unlikely.

The first 3 sublanguages are independent of each other and are named *profiles* [75]. Figure 7.3 illustrate the relationships between the OWL 2 sublanguages.

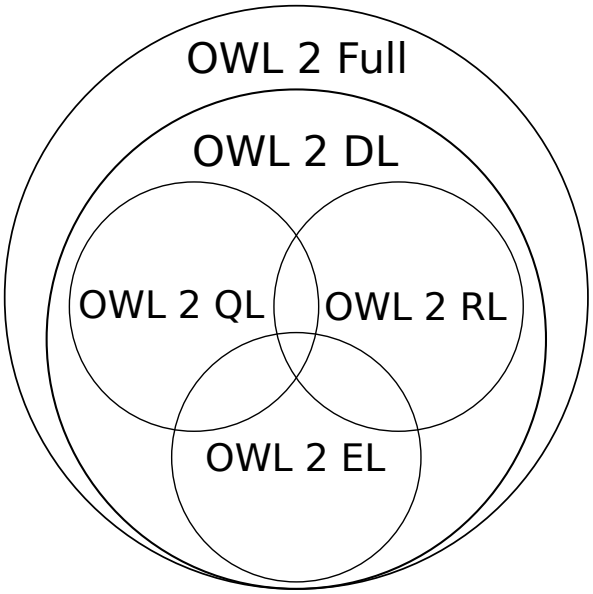


Figure 7.3: OWL 2 sublanguages.

7.6.3 Tools for OWL

The continuous increase in range and sophistication of tools and infrastructure for OWL enabled the use of the OWL technology not only in the field of Semantic Web, but also as a language for ontology development in several other fields: biology, geology, military defence, government and medical science. Examples of these tools are:

- Editors and development environments: Protégé [76], Swoop [77].
- Automatic Reasoners: Hermit [78, 79], Fact++ [80, 81], Pellet[82].[9]
- APIs: OWL API [83, 84], Jena [85] (Both for Java).

7.7 Conclusions

In this chapter we provided a review of Description Logics (DLs) and OWL. FOL is not decidable, hence the interest in identifying decidable fragments (DLs) to be used

[9] A complete list of available reasoners for OWL can be found at `http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/`.

to representing ontologies. Since Many DLs are fragments of FOL, it is possible to map a DL expression into a FOL formula without losing the original "meaning". OWL is a logic language for the Semantic Web based on description logics. In particular OWL 2 DL is based on $\mathcal{SROIQ}(\mathbf{D})$, which is decidable. For a deeper introduction to description logics we refer to [52, 53, 54]. Instead for further information about Semantic Web and OWL syntax we refer to the W3C World Wide Web Consortium online site [55].

The next chapter will discuss how to reason over description logics.

Chapter 8

Reasoning in Description Logics

This chapter illustrates the reasoning problems in Description Logics and techniques for solving them. The reasoning methods here discussed will come in handy when we present the approaches to exact probabilistic inference (Chapter 11).

After an introduction to reasoning problems in Section 8.1. Section 8.2 illustrates some techniques for standard and non-standard reasoning in DLs, in particular it describes Pellet's tableau algorithm and approaches to obtain, given a query, all the explanations and the pinpointing formula. Section 8.3 draws conclusions.

8.1 Reasoning Problems

As mentioned before, a knowledge base is used to store information about the application domain, but the purpose of a knowledge representation system goes beyond storing concept definitions and assertions. Besides the *explicit* knowledge contained in the KB, we want to extract *implicit* knowledge, for example, if we have the two axioms: $\mathsf{jerry} : \mathsf{Person}$ and $\mathsf{Person} \equiv \neg\mathsf{Dinosaur}$, one can conclude that jerry doesn't belongs to the class $\mathsf{Dinosaur}$, even though this knowledge is not explicitly stated as an assertion. *Inference algorithms* have the objective of extracting such implicit knowledge. Let $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ be a DL knowledge base, where $\mathcal{T}$ is the TBox, $\mathcal{R}$ is the RBox and $\mathcal{A}$ the ABox. The standard reasoning tasks are:

Concept satisfiability A concept C is satisfiable with respect to $\mathcal{K}$ iff there exists a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^{\mathcal{I}} \neq \emptyset$. In this case we say that $\mathcal{K} \models C$.

Concept subsumption A concept C is subsumed by a concept D with respect to $\mathcal{K}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$. In this case we say that $\mathcal{K} \models C \sqsubseteq D$.

Concept equivalence Two concepts C and D are equivalent if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every model $\mathcal{I}$. In this case we write $C \equiv D$ or $\mathcal{I} \models C \equiv D$.

Concept disjointness Two concepts C and D are disjoint if $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ for every model $\mathcal{I}$.

Instance checking Finding out whether a given individual a belongs to a given concept C, i.e. $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$.

Instance retrieval Find all instances of a given concept.

Knowledge base consistency A knowledge base $\mathcal{K}$ is *consistent* iff it has a model $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{K}$ (see Definition 7.5), i.e. it is satisfiable.

Classification To compute the subsumption relation between each pair of concepts in $\mathcal{K}$ in order to create the complete concept hierarchy.

Realization To find the most specific concept to which an individuals belongs. The most specific concepts of an individuals are the lowest concepts in the hierarchy to which an individual belongs (i.e. the direct types of an individual). Thus realization is performed after classification.

All of the above reasoning problems can be reduced to KB (in)consistency.

Proposition 8.1 Reduction to KB Inconsistency
Let $\mathcal{K}$ be a DL KB and x a new individual not appearing in $\mathcal{K}$.

Concept satisfiability $\mathcal{K} \models C$ iff it is not the case that $\mathcal{K} \cup \{x : C\}$ is inconsistent, i.e. it is unsatisfiable.

Concept subsumption $\mathcal{K} \models C \sqsubseteq D$ iff $\mathcal{K} \cup \{x : (C \sqcap \neg D\})$ is inconsistent.

Concept equivalence $\mathcal{K} \models C \equiv D$ iff $\mathcal{K} \models C \sqsubseteq D$ and $\mathcal{K} \models D \sqsubseteq C$.

Concept disjointness $\mathcal{K} \models C \sqcap D \sqsubseteq \bot$ iff $\mathcal{K} \cup \{x : (C \sqcap D)\}$ is inconsistent.

Instance check $\mathcal{K} \models a : C$ iff $\mathcal{K} \cup \{a : \neg C\}$ is inconsistent.

Instance retrieval We have to check for every individual a in $\mathcal{K}$ if $\mathcal{K} \models a : C$.

Classification for every pair of concepts C and D in $\mathcal{K}$ check whether $\mathcal{K} \models C \sqsubseteq D$.

□

Other reasoning tasks, called non-standard, have a somewhat different goal:

Induction Inductive approaches usually take a part of intensional knowledge and try to generalize them by generating hypotheses expressed as axioms or complex concepts.

Abduction Abductive reasoning services is useful in ontology engineering when a desired consequence (say E) is not a consequence of the knowledge base $\mathcal{K}$ and the ontology engineer wants to determine what information $\mathcal{K}'$ is missing, such that $\mathcal{K} \cup \mathcal{K}' \models E$.

Explanation finding The goal is to give an explanation of why some axiom is entailed by the knowledge base. Formally, a justification for the entailment is a knowledge base $\mathcal{K}' \subset \mathcal{K}$ such that $\mathcal{K}' \models E$. There might be more than one justification for an entailment. This reasoning task is also called **axiom pinpointing**.

Pinpointing formula extraction Given a query Q and a KB $\mathcal{K}$, the *pinpointing formula* is a monotone Boolean formula ψ built using boolean variables associated to each axiom in $\mathcal{K}$ and only the conjunction and disjunction connectives[1]. Every valuation of ψ must correspond to a knowledge base $\mathcal{K}_\psi \subset \mathcal{K}$ such that $\mathcal{K}_\psi \models Q$.

8.1.1 Closed vs Open World Assumption

Description Logics do not adopt the Closed World Assumption (CWA), but like FOL, make the Open World Assumption (OWA). Missing information is treated as unknown.

Example 8.1.1
Let us consider the following example:

$$\mathsf{Father} \equiv \mathsf{Male} \sqcap \exists \mathsf{hasChild}.\top$$
$$\{\mathsf{markus}, \mathsf{anna}\} : \mathsf{hasChild}$$
$$\mathsf{markus} : \mathsf{Father}$$
$$\mathsf{anna} : \mathsf{Female}$$
$$\mathsf{stephan} : \mathsf{Male}$$

The query is: is stephan son of markus (markus,stephan) : hasChild*?*

- *In DLs due to the OWA the answer is: don't know.*
- *In databases and in logic programming we usually have the closed world assumption (see Section 4.4). Missing information is assumed to be false. Hence, in these contexts with the CWA, the answer is: no.*

8.2 Reasoning Techniques

To solve the reasoning problems introduced in Section 8.1, several reasoning techniques have been proposed. These includes automata based approaches [86], resolution based approaches [87, 88] and structural approaches [89]. The most widely used technique, however, is the *tableau* approach. Indeed the vast majority of state-of-the-art OWL reasoners, such as Pellet [82], FaCT++ [81, 80] and Hermit [78, 79], use a tableau algorithm.

In the next subsections we describe the tableau algorithm and, in particular, the rules used by the Pellet reasoner. The aim of this chapter is to illustrate how to perform two non-standard reasoning tasks:

- **explanation finding**, also known as **axiom pinpointing**; and
- **pinpointing formula extraction**.

The approaches described below are based on [90, 91, 92].

[1] In monotone Boolean formulas the *not* connective is not used.

8.2.1 Pellet

Pellet is a complete OWL 2 reasoner, that covers all the OWL 2 DL constructs including inverse and transitive properties, cardinality restrictions, datatype reasoning for an extensive set of built-ins as well as user defined simple XML schema datatypes, enumerated classes (nominals) and instance assertions.

This practical OWL reasoner provides the "standard" set of DL inference services, namely consistency checking of a KB, concept satisfiability, classification, realization. Pellet reduces them all to KB consistency checking. These services can be accessed by querying the reasoner. Pellet also supports some less standard services such as *axiom pinpointing/explanation finding*.

The core of the system is the tableau reasoner, which has only one functionality: checking the consistency/satisfiability of a KB. According to the DL model-theoretic semantics, a KB is consistent if there is an interpretation that satisfies all the facts and axioms in the KB i.e., a model of the KB. The tableau reasoner searches for such a model.

Pellet is written in Java. It is used in a number of projects, from pure research to industrial ones. Until version 2.3.0 Pellet was an open source project, but the later versions are closed source. However there are several open source forks of the original project, one of them is Openllet[2].

8.2.2 Tableau Algorithm

In the following we describe the tableau algorithm used by Pellet, shown in Algorithm 8.1.

The idea behind the algorithm is essentially to try to construct a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$. If we find a model, then $\mathcal{K}$ is obviously satisfiable, otherwise it is inconsistent. It does this in an organized way by starting from the concrete situation described in $\mathcal{A}$, and explicating additional constraints on the model that are implied by the axioms in $\mathcal{K}$.

The algorithm works on data structures called **tableaux**, which are **completion graphs**. Formally, a completion graph is a tuple $G = (V, E, \mathcal{L}, \neq)$ in which (V, E) is a directed graph. Each node $a \in V$ is labelled with a set of concepts $\mathcal{L}(a)$, and each edge $\langle a, b\rangle$ is labelled with a set of role names $\mathcal{L}(\langle a, b\rangle)$. The binary predicate $\neq$ is used to specify the inequalities between nodes. A tableau can also be seen as an ABox $\mathcal{A}$, where the nodes are individuals annotated with the concepts they belong to and the edges are annotated with the roles that relate the connected individuals.

Function TABLEAU in Algorithm 5 takes as input a query axiom Q and a KB $\mathcal{K}$. In line 5 Q is **negated** for unsatisfiability, i.e. it is converted into an assertional axiom to be used for KB inconsistency checking, as explained in Proposition 8.1. For instance, if our query is $Q = a : C$ then it is transformed into $\alpha_Q = a : \neg C$. α_Q is called the *assertional negation* of Q. The algorithm keeps a set T of completion graphs. T is initialized with a single completion graph G_0 that contains a node for each individual a in the knowledge base, labeled with the nominal $\{a\}$ plus all concept C such that $a : C \in \mathcal{A}$ (i.e. $\mathcal{L}(a) = \{C \mid a : C \in \mathcal{A}\} \cup \{\{a\}\}$), and an edge $\langle a, b\rangle$ labeled with R for

[2]GitHub repository: `https://github.com/Galigator/openllet`.

Algorithm 8.1 Tableau algorithm executed by Pellet.

```
1: function TABLEAU(Q, K)
2:     Input: Q (the query axiom)
3:     Input: K (the knowledge base)
4:     Output: S (a set of axioms) or null
5:     α_Q = UNSAT(Q)          ▷ UNSAT(Q) converts Q into an assertional axiom to be tested for
   inconsistency
6:     K = K ∪ {α_Q}
7:     Let G_0 be an initial completion graph from K
8:     T ← {G_0}
9:     repeat
10:        Select a rule r applicable to a clash-free graph G from T
11:        T ← T \ {G}
12:        Let G = {G'_1, ..., G'_n} be the result of applying r to G
13:        T ← T ∪ G
14:    until All graphs in T have a clash or no rule is applicable
15:    if All graphs in T have a clash then
16:        S ← ∅
17:        for all G ∈ T do
18:            let s_G the result of τ for the clash of G
19:            S ← S ∪ s_G
20:        end for
21:        S ← S \ {α_Q}
22:        return S
23:    else
24:        return null
25:    end if
26: end function
```

each assertion $(a, b) : R \in \mathcal{A}$ (i.e. $\mathcal{L}(\langle a, b\rangle) = \{R|(a, b) : R \in \mathcal{A}\}$).

The algorithm then applies, at each step, a so-called **expansion rule** to a completition graph $G \in T$: G is removed from T, the rule is applied and the results are inserted back in T. The some of the rules used by Pellet for $\mathcal{SROIQ}(\mathbf{D})$ DL are shown in Figure 8.1. For example, if $C_1 \sqcap C_2 \in \mathcal{L}(a)$, and either $C_1 \notin \mathcal{L}(a)$ or $C_2 \notin \mathcal{L}(a)$, then the rule $\rightarrow \sqcap$ adds both C_1 and C_2 to $\mathcal{L}(a)$, because the individual a must be an instance of both C_1 and C_2.
Rules can be deterministic or non-deterministic. The first replace G with a single graph while the latter replace G with a set of graphs. For example, if the disjunction $C_1 \sqcup C_2$ is present in the label of a node and neither $C_1 \in \mathcal{L}(a)$ nor $C_2 \in \mathcal{L}(a)$, the rule $\rightarrow \sqcup$ generates two graphs, one in which C_1 is added to $\mathcal{L}(a)$ and another in which C_2 is added to $\mathcal{L}(a)$.

An *event* during the execution of the algorithm can be:

- $Add(C, a)$, the addition of a concept C to $\mathcal{L}(a)$.
- $Add(R, \langle a, b\rangle)$, the addition of a role R to $\mathcal{L}(\langle a, b\rangle)$.
- $Merge(a, b)$, the merging of the nodes a, b. When one node b is merged into another node a, $\mathcal{L}(b)$ is added into $\mathcal{L}(a)$, all the edges leading to b are "moved" so that they lead to a, and all the edges leading from b to nominal nodes are "moved" so that they lead from a to the same nominal nodes; then b and the

Deterministic rules:

→ *unfold* (∗): **if** $A \in \mathcal{L}(a)$, A atomic and $(A \sqsubseteq D) \in K$, **then**
if D $\notin \mathcal{L}(a)$, **then**
$Add(D, a)$, $\tau(D, a) := (\tau(A, a) \cup \{A \sqsubseteq D\})$

→ *CE* (∗): **if** $(C \sqsubseteq D) \in K$, with C not atomic, a not blocked, **then**
if $(\neg C \sqcup D) \notin \mathcal{L}(a)$, **then**
$Add((\neg C \sqcup D), a)$, $\tau((\neg C \sqcup D), a) := \{C \sqsubseteq D\}$

→ ⊓ (∗): **if** $(C_1 \sqcap C_2) \in \mathcal{L}(a)$, a is not indirectly blocked, **then**
if $\{C_1, C_2\} \not\subseteq \mathcal{L}(a)$, **then**
$Add(\{C_1, C_2\}, a)$, $\tau(C_i, a) := \tau((C_1 \sqcap C_2), a)$

→ ∃ (∗): **if** $\exists S.C \in \mathcal{L}(a)$, a is not blocked, **then**
if a has no S-neighbor b with $C \in \mathcal{L}(b)$, **then**
create new node b, $Add(S, \langle a, b\rangle)$, $Add(C, b)$
$\tau(C, b) := \tau((\exists S.C), a)$, $\tau(S, \langle a, b\rangle) := \tau((\exists S.C), a)$

→ ∀ (∗): **if** $\forall(S.C) \in \mathcal{L}(a)$, a is not indirectly blocked and
there is an S-neighbor b of a, **then**
if $C \notin \mathcal{L}(b)$, **then**
$Add(C, b)$, $\tau(C, b) := \tau((\forall S.C), a) \cup \tau(S, \langle a, b\rangle)$

→ $\forall^+$ (∗): **if** $\forall(S.C) \in \mathcal{L}(a)$, a is not indirectly blocked and
there is an R-neighbor b of a, $Trans(R)$ and $R \sqsubseteq S$, **then**
if $\forall R.C \notin \mathcal{L}(b)$, **then**
$Add(\forall R.C, b)$,
$\tau((\forall R.C), b) := \tau((\forall S.C), a) \cup \tau(R, \langle a, b\rangle) \cup \{Trans(R)\} \cup \{R \sqsubseteq S\}$

→≥ (∗): **if** $(\geq nS) \in \mathcal{L}(a)$, a is not blocked, **then**
if there are no n safe S-neighbors $b_1, ..., b_n$ of a with $b_i \neq b_j$, **then**
create n new nodes $b_1, ..., b_n$, $Add(S, \langle a, b_i\rangle); \neq(b_i, b_j)$
$\tau(S, \langle a, b_i\rangle) := \tau((\geq nS), a)$, $\tau(\neq(b_i, b_j)) := \tau((\geq nS), a)$

→ O (∗): **if**, $\{o\} \in \mathcal{L}(a) \cap \mathcal{L}(b)$ and not $a \neq b$, **then** $Merge(a, b)$
$\tau(Merge(a, b)) := \tau(\{o\}, a) \cup \tau(\{o\}, b)$
For each concept C_i in $\mathcal{L}(a)$ **then**
$\tau(C_i, b) := \tau(C_i, a) \cup \tau(Merge(a, b))$
(similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(b)$)

Non-deterministic rules:

→ ⊔ (∗): **if** $(C_1 \sqcup C_2) \in \mathcal{L}(a)$, a is not indirectly blocked, **then**
if $\{C_1, C_2\} \cap \mathcal{L}(a) = \emptyset$, **then**
Generate graphs $G_i := G$ for each $i \in \{1, 2\}$
$Add(C_i, a)$, $\tau(C_i, a) := \tau((C_1 \sqcup C_2), a)$ in G_i for each $i \in \{1, 2\}$

→≤ (∗): **if** $(\leq nS) \in \mathcal{L}(a)$, a is not indirectly blocked,
and there are m S-neighbors $b_1, ..., b_m$ of a with $m > n$, **then**
For each possible pair b_i, b_j, $1 \leq i, j \leq m; i \neq j$ **then**
Generate a graph G'
$\tau(Merge(b_i, b_j)) := \tau((\leq nS), a) \cup \tau(S, \langle a, b_1\rangle) ... \cup \tau(S, \langle a, b_m\rangle)$
if b_j is a nominal node, **then** $Merge(b_i, b_j)$ in G',
else if b_i is a nominal node or ancestor of b_j, **then** $Merge(b_j, b_i)$
else $Merge(b_i, b_j)$ in G'
if b_i is merged into b_j, **then** for each concept C_i in $\mathcal{L}(b_i)$,
$\tau(C_i, b_j) := \tau(C_i, b_i) \cup \tau(Merge(b_i, b_j))$
(similarly for roles merged, and correspondingly for concepts in b_j
if merged into b_i)

→ NN : **if** $(\leq nS.C) \in \mathcal{L}(a)$, a nominal node, b blockable S-predecessor of a and
there is no m s.t. $1 \leq m \leq n$, $(\leq mS.C) \in \mathcal{L}(a)$ and
there exist m nominal S-neighbors $z_1, \ldots, z_m$ of a s.t.
$C \in \mathcal{L}(z_i)$ and $z_i \neq z_j$ for all $1 \leq i \leq j \leq m$, **then**
For each k, $1 \leq k \leq n$, **then**
Generate a graph G_k
$Add(\leq kS.C, a)$, $\tau((\leq kS.C), a) := \tau((\leq nS.C), a) \cup \tau(S, \langle b, a\rangle)$
create $b_1, \ldots, b_k$, *Add* $b_i \neq b_j$ for $1 \leq i \leq j \leq k$, $\tau(\neq(b_i, b_j)) := \tau((nS.C), a) \cup \tau(S, \langle b, a\rangle)$,
$Add(S, \langle a, b_i\rangle)$, $Add(\{o_i\}, b_i)$, where o_i are new nominals,
$\tau(S, \langle a, b_i\rangle) := \tau((\leq nS.C), a) \cup \tau(S, \langle b, a\rangle)$, $\tau(\{o_i\}, b_i) := \tau((\leq nS.C), a) \cup \tau(S, \langle b, a\rangle)$

Figure 8.1: Some Pellet tableau expansion rules for $\mathcal{SROIQ}(\mathbf{D})$; the subset of rules marked by (∗) are relevant for $\mathcal{SHIQ}$.

blockable sub-trees below b are *pruned*, i.e. removed, from the tableau. Merge and prune operations are described in detail in [93, 61].

- $\neq (a, b)$, the addition of the inequality $a \neq b$ to the relation $\neq$.
- $Report(g)$, the detection of a clash g.

We use $\mathcal{E}$ to denote the set of events recorded during the execution of the algorithm. A clash is either:

- a couple (C, a) where both C and $\neg C$ are present in the label of a node (there is an inconsistency), i.e. $\{C, \neg C\} \subseteq \mathcal{L}(a)$;
- a couple $(Merge(a, b), \neq (a, b))$, where the events $Merge(a, b)$ and $\neq (a, b)$ belong to $\mathcal{E}$.

The algorithm stops applying rules to G if it encounters a clash. In this case, the completion graph G contains an inconsistency, and thus does not represent a model. If no more expansion rules can be applied to the completion graph G and there are no clashes, then G represent a model. Once every completion graph in T contains a clash or no more expansion rules can be applied to it, then the algorithm *terminates*. If all the completion graphs in the final set T contain a clash, then the algorithm returns "$\mathcal{K}$ is unsatisfiable", i.e. no model can be found. Otherwise, all the clash-free completion graphs in T represents a model for $\mathcal{K}$ and the algorithm returns "$\mathcal{K}$ is consistent". The tableau algorithm is known to be *sound* and *complete*.

For ensuring the termination of the algorithm, a special condition known as *blocking* [90] is used. In a tableau a node x can be a *nominal* node if its label $\mathcal{L}(x)$ contains a *nominal* or a *blockable* node[3]. If there is an edge $e = \langle x, y \rangle$ then y is a *successor* of x and x is a *predecessor* of y. *Descendant* is the transitive closure of successor while *ancestor* is the transitive closure of predecessor. A node y is called an R-successor of a node x if, for some R' with $R' \sqsubseteq\!\!\!\!* \; R$, $R' \in \mathcal{L}(\langle x, y \rangle)$, where $\sqsubseteq\!\!\!\!*$ is the transitive-reflexive closure of $\sqsubseteq$ on $\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$ and $\mathsf{Inv}(R)$ is a function that returns the inverse of a role R. x is called an R-predecessor of y if y is an $\mathsf{Inv}(R)$-successor of x. A node y is called a neighbour (R-neighbour) of a node x if y is either a successor (R-successor) or a predecessor (R-predecessor) of x.

For a role S and a node x, we define the set of x's S-neighbours with C in their label, $S(x, C)$, as

$$S(x, C) := \{y \mid y \text{ is an } S\text{-neighbour of } x \text{ and } C \in \mathcal{L}(y)\}.$$

An R-neighbor y of x is *safe* if

- x is blockable, or
- x is a nominal node and y is not blocked

Finally, a node x is *blocked* if it has ancestors x_0, y and y_0 such that all the following conditions are true:

[3]Note that, when a new node is added by an expansion rules, this node is blockable.

1. x is a successor of x_0 and y is a successor of y_0,
2. y, x and all nodes on the path from y to x are blockable,
3. $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x_0) = \mathcal{L}(y_0)$,
4. $\mathcal{L}(\langle x_0, x\rangle) = \mathcal{L}(\langle y_0, y\rangle)$.

In this case, we say that y *blocks* x. A node is blocked also if it is blockable and all its predecessors are blocked; if the predecessor of a safe node x is blocked, then we say that x is indirectly blocked.

8.2.3 Explanation finding

Here we discuss the problem of finding covering set of explanations for a given query. This non-standard reasoning service is useful for tracing derivations and debugging ontologies and has been investigated by various authors [94, 90, 95, 96, 97]. Schlobach and Cornet [97] named it **axiom pinpointing**. In particular, the authors of [97] define *minimal axiom sets* or *MinAs* for short.

Definition 8.1 MinA
Let $\mathcal{K}$ be a knowledge base and Q an axiom that follows from it, i.e., $\mathcal{K} \models Q$. We call a set $M \subseteq \mathcal{K}$ a *minimal axiom set* or *MinA* for Q in $\mathcal{K}$ if $M \models Q$ and it is minimal w.r.t. set inclusion. A MinA corresponds to an *explanation* for the query Q. □

The problem of enumerating all MinAs is called MIN-A-ENUM in [97]. ALL-MINAS$(Q, \mathcal{K})$ is the set of all MinAs for query Q in the knowledge base $\mathcal{K}$. We can formally define the MIN-A-ENUM problem as follows

Definition 8.2 MIN-A-ENUM problem
Input: A knowledge base $\mathcal{K}$, and an axiom Q such that $\mathcal{K} \models Q$.
Output: The set ALL-MINAS$(Q, \mathcal{K})$ of all MinAs for Q in $\mathcal{K}$. □

The algorithm for computing a single MinA is shown in Algorithm 8.2. It takes advantage of function TABLEAU (line 5) and of function BLACKBOXPRUNING (line 9). TABLEAU exploits the tableau algorithm presented in Subsection 8.2.2: given a KB $\mathcal{K}$ if our query is $Q = a : C$, the tableau algorithm works by refutation and it tries to prove the inconsistency of $\mathcal{K} \cup \{\alpha_Q\}$, where $\alpha_Q = a : \neg C$. If no model can be build then $\mathcal{K} \models Q$, see Proposition 8.1.

Function TABLEAU

Every time a rule is applied, Pellet update a so-named *tracing function* τ [94, 90, 98], which associates sets of axioms with events in the derivation.

The tracing function τ maps each event $\varepsilon \in \mathcal{E}$ to a fragment of $\mathcal{K}$. For example, $\tau(Add(C, a))$ is the set of axioms needed to explain the event $Add(C, a)$ while $\tau(Add(R, \langle a, b\rangle))$ explains the event $Add(R, \langle a, b\rangle)$. For the sake of brevity we define τ for couples (concept, individual) and (role, couple of individuals) as $\tau(C, a) = \tau(Add(C, a))$ and $\tau(R, \langle a, b\rangle) = \tau(Add(R, \langle a, b\rangle))$ respectively. The function τ is initialized as the empty set for all the elements of its domain except for $\tau(C, a)$ and

Algorithm 8.2 SINGLEMINA algorithm.

```
1: function SINGLEMINA(Q, K)
2:     Input: Q (the query axiom)
3:     Input: K (the knowledge base)
4:     Output: S (a MinA for the Q) or null
5:     S ←TABLEAU(Q, K)
6:     if S = null then
7:         return null
8:     else
9:         return BLACKBOXPRUNING(Q, S)
10:    end if
11: end function
```

$\tau(R, \langle a, b\rangle)$ to which the values $\{a : C\}$ and $\{(a, b) : R\}$ are assigned if $a : C$ and $(a, b) : R$ are in the ABox respectively. The expansion rules (Figure 8.1) add axioms to values of τ.

For a clash of the form (C, a), $\tau(Report(g)) = \tau(Add(C, a)) \cup \tau(Add(\neg C, a))$. Instead, for a clash of the form $(Merge(a, b), \neq (a, b))$, $\tau(Report(g)) = \tau(Merge(a, b)) \cup \tau(\neq (a, b))$.

If $g_1, ..., g_n$ are the clashes, one for each of the elements of the final set of tableaux and $\tau(Report(g_i)) = s_{g_i}$, the output of the algorithm TABLEAU is $S = \bigcup_{i\in\{1,...,n\}} s_{g_i} \setminus \{\alpha_Q\}$ where α_Q is the *assertional negation* of our initial query Q. However, this set may be redundant because additional axioms may also be included in τ, e.g., during the $\rightarrow\leq$ rule, where axioms responsible for each of the successor edges are considered.

Function BLACKBOXPRUNING

The set S, returned by Function TABLEAU is pruned using a black-box approach [90] called BLACKBOXPRUNING and shown in Algorithm 8.3. Given a query axiom Q, this algorithm executes a loop on S: in each iteration it removes an axiom E from S and checks whether $S \models Q$ by means of TABLEAU. If the query Q is not entailed, the axiom E is reinserted into S as E is responsible for the entailment of the query Q. Vice-versa, if the query still remains entailed, the removed axiom E is irrelevant and is not reinserted in S. Once all axioms in S have been tested the process terminates and returns S. Thus the algorithm for computing a single MinA SINGLEMINA, shown in Algorithm 8.2, first executes TABLEAU and then BLACKBOXPRUNING.

The output S of SINGLEMINA is guaranteed to be a MinA, as established by Theorem 8.1, where ALL-MINAS$(Q, \mathcal{K})$ stands for the set of all MinAs for Q.

Theorem 8.1 [90]
Let Q be a query and let S be the output of the algorithm SINGLEMINA *with input Q and $\mathcal{K}$, then $S \in$* ALL-MINAS$(Q, \mathcal{K})$.

Hitting Set Algorithm

SINGLEMINA returns a single MinA. To compute all MinAs, Pellet uses Reiter's *hitting set algorithm* [99]. In [99], Reiter developed a general theory of diagnosis where a system to be diagnosed is a pair $(SD, COMP)$ where SD is a set of first-order sentences which

Algorithm 8.3 Black-Box pruning algorithm.

```
1: function BlackBoxPruning(C, S)
2:     Input: Q (the query axiom)
3:     Input: S (the set of axioms to be pruned)
4:     Output: S (the pruned set of axioms)
5:     for all axiom E ∈ S do
6:         S ← S − {E}
7:         if Tableau(Q, S) = null then
8:             S ← S ∪ {E}
9:         end if
10:    end for
11:    return S
12: end function
```

describe the system and $COMP$ is a finite set of components. A set of observation OBS is then associated with the system. An observation is finite set of first-order sentences which describe the behavior of the system. In a system there can be some components that are abnormal, i.e. components whose behavior is not correct. Reiter defined a *diagnosis* for a system as a minimal set $\Delta \subseteq COMP$ such that

$$SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|(c) \in COMP - \Delta\}$$

is consistent, where AB is a predicate that indicates whether a component is abnormal. This means that a diagnosis is the minimal set of faulty components which combined with the other components, which are normal, make the system consistent. A diagnosis can be defined in terms of *conflict sets*, that are sets $\{c_1, ...c_n\} \subseteq COMP$ s.t.

$$SD \cup OBS \cup \{\neg AB(c_1), ..., \neg AB(c_n)\}$$

is inconsistent. A conflict set is *minimal* iff no proper subset of it is a conflict set for the observed system. In this characterization, a diagnosis Δ is a *minimal* set s.t. $COMP - \Delta$ is not a conflict set for the system.

Let us consider a *universal set* U and a set of *conflict sets* $CS \subseteq \mathcal{P}(U)$, where $\mathcal{P}$ denotes the powerset operator. The set $HS \subseteq U$ is a *hitting set* for CS if each $S_i \in CS$ contains at least one element of HS, i.e. if $S_i \cap HS \neq \emptyset$ for all $1 \leq i \leq n$ (in other words, HS 'hits' or intersects each set in CS). HS is a *minimal hitting set* for CS if HS is a hitting set for CS and no $HS' \subset HS$ is a hitting set for CS.

The *hitting set problem* with input CS, U is to compute all the minimal hitting sets for CS. The set of all minimal conflict sets, which correspond to the explanations for inconsistency, can be found by exploiting an algorithm that generates minimal hitting sets [90, 95].

Reiter's algorithm [99] constructs a labeled tree called **hitting set tree (HST)** as follows.

In an HST, a node v is labeled with OK, or with a set $\mathcal{L}(v) \in CS$ and an edge e is labeled with an element of U. The label of a node v (edge e) is denoted as $\mathcal{L}(v)$ ($\mathcal{L}(e)$). Let T be an HST.

- If v is the root of T, if CS is empty, it is labeled with OK, i.e. $\mathcal{L}(v) \leftarrow OK$. Otherwise, it is labeled with a set $S \in CS$ ($\mathcal{L}(v) \leftarrow S$).

- If v is a node of T, we define $H(v)$ as the set of edge labels on the path from the root of T to node v. If v is labeled with OK, it is a leaf. Otherwise, v is labeled with a set $S \in CS$ ($\mathcal{L}(v) \leftarrow S$) and, for each element $E \in \mathcal{L}(v)$, v has a successor w connected to v by an edge with E in its label. The label for w is a set $S \in CS$ such that $S \cap H(w) = \emptyset$ if S exists, otherwise w is labeled with OK.

Reiter showed that

- if $\mathcal{L}(v) = OK$, then $H(v)$ is a hitting set for CS;
- each minimal hitting set for CS is $H(v)$ for some node v with label OK.

Example 8.2.1
(From [99]) Let us consider the following set of conflict sets

$$CS = \{\{2,4,5\},\{1,2,3\},\{1,3,5\},\{2,4,6\},\{2,4\},\{2,3,5\},\{1,6\}\}$$

Figure 8.2 shows an HST for CS.

We can notice that in the HST in Figure 8.2 we could have pruned several paths. For instance $H(n_6) = H(n_8)$. This means that the subtrees rooted at n_6 and n_8 could be identically generated, i.e. the subtree rooted at n_8 is redundant. In addition, $H(n_3) = \{1,2\}$ is a minimal hitting set for CS. Therefore, it is not possible that any other node v of the tree such that $H(n_3) \subseteq H(v)$ can define a smaller hitting set than $H(n_3)$. Moreover if $S \in CS$ and $S' \in CS$ with $S \subset S'$, then $CS \setminus S'$ has the same minimal hitting sets as CS, in Figure 8.2 we have that $\mathcal{L}(n_{10}) = \{2,4\} \subset \{2,4,5\} = \mathcal{L}(n_0)$.

From these observations Reiter defined three rules for HST pruning.

Proposition 8.2 HST pruning
An HST can be pruned by following these rules without losing any minimal hitting set.

1. If a node v is labeled with OK and the current node w is such that $H(v) \subseteq H(w)$, then *close* the node w by labeling it with an X and make it a leaf: $\mathcal{L}(w) \leftarrow X$. In fact once a hitting set path is found any superset of that path is guaranteed to be a hitting set as well.

2. If there exists a node v such that $H(v) = H(w)$, where w is the current node, then label w with X.

3. If there exist two nodes v and w such that $\mathcal{L}(w) \subset \mathcal{L}(v)$, then for each $E \in \mathcal{L}(v) \setminus \mathcal{L}(w)$ mark as redundant the edge from node v labeled by E. A redundant edge, together with the subtree beneath it, may be removed from the HST while preserving the property that the resulting pruned HST will yield all minimal hitting sets for CS.

□

In our case, the universal set U corresponds to the set of all axioms in the KB, and an explanation (for a particular KB inconsistency) corresponds to a single conflict set $S \in CS$ [90, 95]. While the main aim of Reiter's approach was to find all minimal

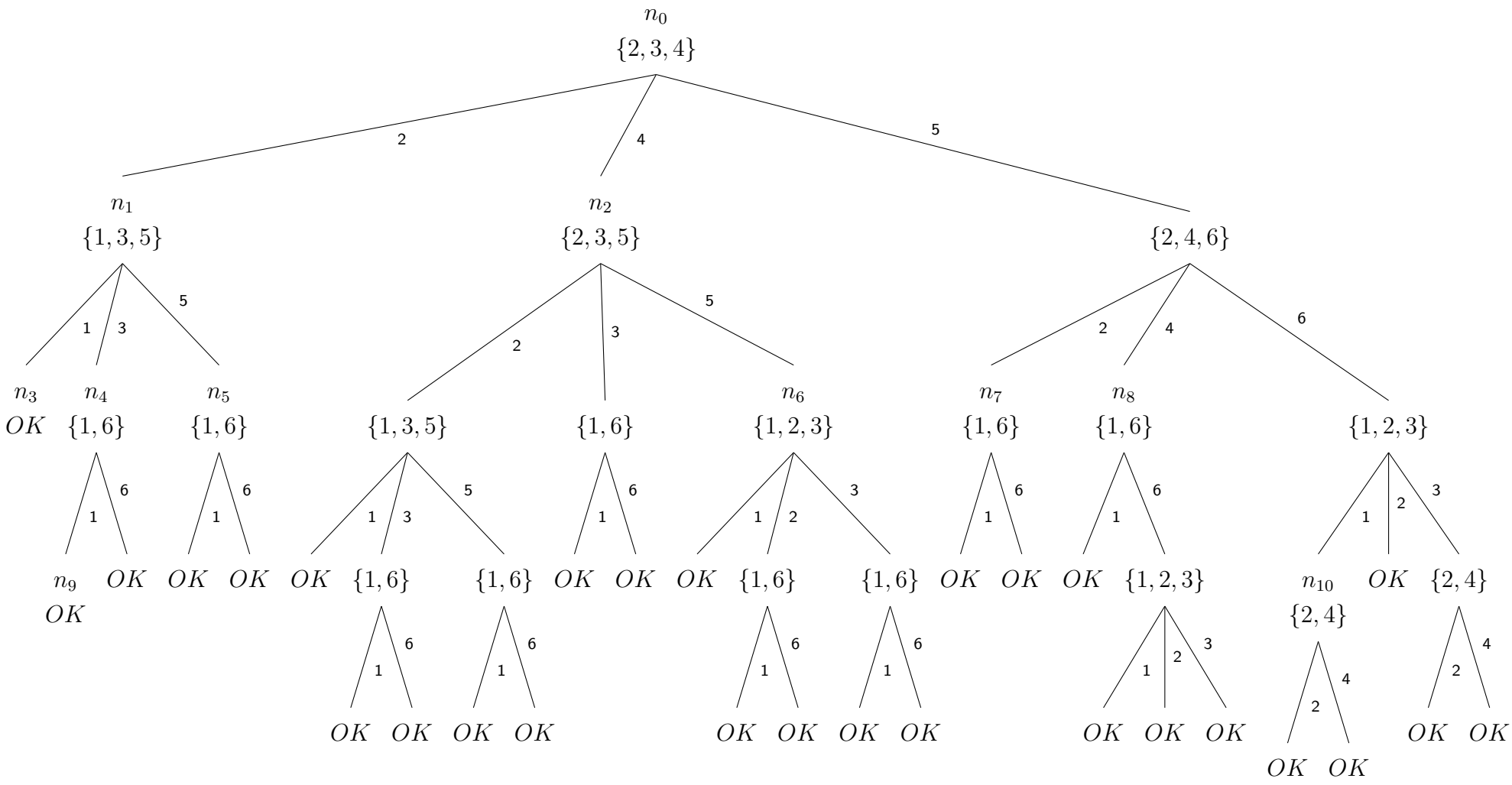


Figure 8.2: An HST for $CS = \{\{2,4,5\},\{1,2,3\},\{1,3,5\},\{2,4,6\},\{2,4\},\{2,3,5\},\{1,6\}\}$

hitting sets of a set of conflict sets CS, due to the duality of the algorithm, it can also be used to find all conflict sets, which are the explanations in our case.

The algorithm for HST construction is shown in Algorithm 8.4, which is based on the algorithms proposed in [90, 95]. Given a query axiom Q, the algorithm starts by generating the MinA S by invoking SingleMinA with inputs the query axiom Q and the KB $\mathcal{K}$. If $S \neq null$, i.e. $\mathcal{K} \models Q$ (line 16), then S represent a new explanation for Q. If v is $null$, which means that we are creating the root node, it initializes an HST $\mathbf{T} = (V, E, \mathcal{L})$ (line 14). Then it labels v with S and, for each axiom $E \in S$, a new node w and a new edge $\langle v, w \rangle$ labeled with the axiom E are added in the tree, removes E from $\mathcal{K}$, generating a new knowledge base $\mathcal{K}' = \mathcal{K} - \{E\}$, and function HittingSetTree is recursively invoked for the newly generated node w. When $\mathcal{K} \not\models Q$[4] (line 25), the algorithm labels the node v with OK and makes it a leaf.

Lines 8-10 are used to satisfy the first and the second pruning rule of Proposition 8.2: if the path of the current node is a superset of an previously found hitting set or the path of the current node is a path of a previously generated node, then the algorithm labels the current node with a X and makes it a leaf. We don't need to take into account the third rule of Proposition 8.2. In fact, thanks to function BlackBoxPruning (Algorithm 8.3) that assures that the found explanation is minimal, it can never happen that there exist two nodes v and w such that $\mathcal{L}(w) \subset \mathcal{L}(v)$.

When the HST is fully built, all leaves of the tree are labeled with OK or X. The set All-MinAs$(Q, \mathcal{K})$ for the query Q is represented by all distinct non leaf nodes of the tree.

Example 8.2.2
(From [90]) In order to describe the algorithm, let us consider a knowledge base $\mathcal{K}$ with ten axioms and a query Q. For the purpose of the example, we denote the axioms in $\mathcal{K}$ with natural numbers. Suppose All-MinAs$(Q, \mathcal{K})$ *is*

$$\text{All-MinAs}(Q, \mathcal{K}) = \{\{1, 2, 3\}, \{1, 5\}, \{2, 3, 4\}, \{4, 7\}, \{3, 5, 6\}, \{2, 7\}\}$$

Figure 8.3 (taken from [3]) shows the HST that is generated by the algorithm. It starts by computing a single explanation that returns $S = \{2, 3, 4\}$. In the next step, it initializes an HST in which the root node is labeled with S. Then, the algorithm selects an arbitrary axiom in S, say 2, generates a new node w and a new edge $\langle v, w \rangle$ with axiom 2 as its label. The algorithm tests whether $\mathcal{K} - \{2\} \models Q$. If Q is entailed, as in our case, we obtain a new explanation for Q w.r.t. $\mathcal{K} - \{2\}$, say $\{1, 5\}$. We add this set to CS and also assign it to the label of the new node w.

The algorithm repeats this process, i.e. adding a node, removing an axiom and checking entailment, until the entailment test turns negative, in which case we mark the new node with OK.

The correctness of this approach relies on the following key observations:

1. If a node is not a leaf of HST, then its label is an element of the set All-MinAs$(Q, \mathcal{K})$

[4] I.e. $\mathcal{K} \cup \{\alpha_Q\}$ is consistent, where α_Q is the assertional negation of Q

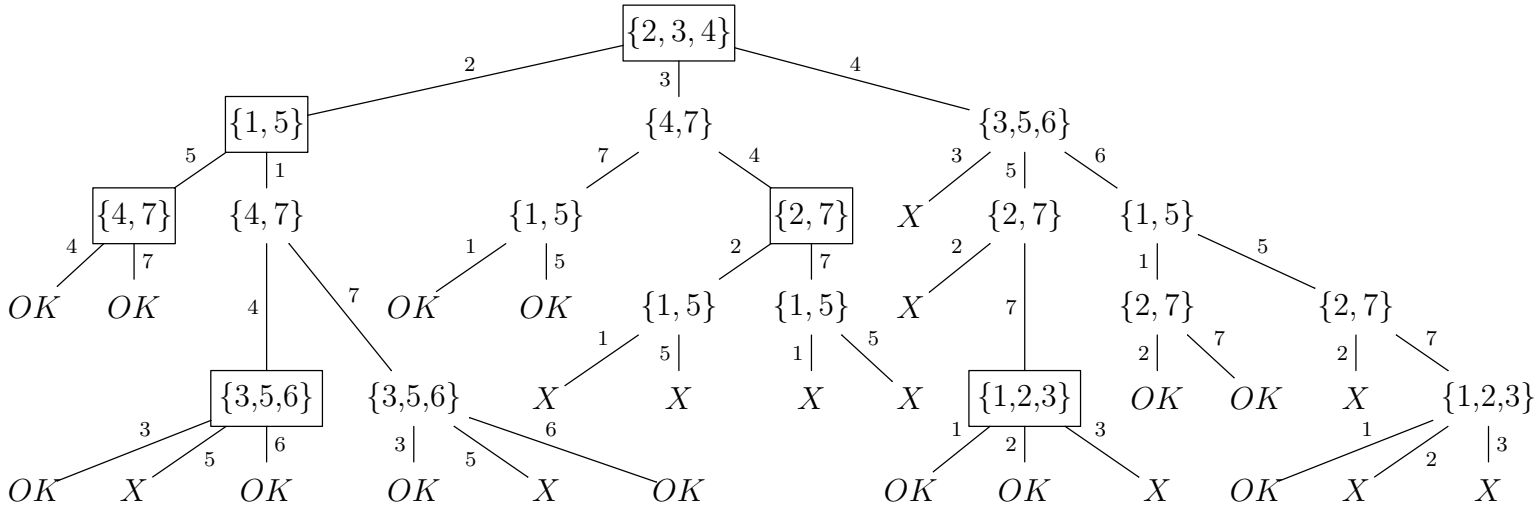


Figure 8.3: Representation of the execution of the hitting set algorithm for finding All-MinAs$(Q, \mathcal{K})$. In the graph, boxed nodes are the set of distinct nodes representing a set in All-MinAs$(Q, \mathcal{K})$.

Algorithm 8.4 Hitting Set Tree Algorithm.

```
1: procedure HittingSetTree(Q, K, CS, HS, w, E, p)
2:     Input: Q (the query axiom)
3:     Input: K (the knowledge base)
4:     Input/Output: CS (a set of explanations, initially empty)
5:     Input/Output: HS (a set of Hitting Sets, initially empty)
6:     Input: v (the last node added to the Hitting Set Tree, initially null)
7:     Input: p (the current edge path, initially empty)
8:     if there exists a set h ∈ HS s.t. h ⊆ p or there exists a node n s.t. H(n) = p then
9:         L(v) ← X
10:        return
11:    else
12:        S ←SingleMinA(Q, K)                          ▷ it checks whether K ⊨ Q
13:        if v = null then
14:            initialization
15:        end if
16:        if S ≠ null then                                          ▷ i.e. K ⊨ Q
17:            CS ← CS ∪ {S}
18:            set L(v) ← S
19:            loop for each axiom E ∈ L(v)
20:                create a new node w and an edge e = ⟨v, w⟩ with L(e) = E
21:                p ← p ∪ L(e)                          ▷ i.e. H(w) ← H(v) ∪ L(e)
22:                K' ← K − {E}
23:                HittingSetTree(Q, K', CS, HS, w, p)
24:            end loop
25:        else                                                      ▷ i.e. K ⊭ Q
26:            L(v) ← OK
27:            HS ← HS ∪ p
28:        end if
29:    end if
30: end procedure
```

2. If one takes the union of the labels of the edges in any path from the root of HST to a leaf node marked with OK, then a hitting set for All-MinAs$(Q, \mathcal{K})$ w.r.t. $\mathcal{K}$ is obtained. In fact, all the minimal hitting sets for All-MinAs$(Q, \mathcal{K})$ are obtained when all the paths from the root to a leaf in HST are considered.

Formally, the correctness and completeness of the hitting set algorithm is given by the following theorem.

Theorem 8.2 [90]
Let Q be a query and $\mathcal{K}$ be a DL KB and let ExpHST$(Q, \mathcal{K})$ *be the set of explanations returned by the hitting set tree algorithm, then* ExpHST$(Q, \mathcal{K})$ *is equal to the set of all explanations of the query Q, so*

$$\text{ExpHST}(Q, \mathcal{K}) \equiv \text{All-MinAs}(Q, \mathcal{K})$$

Reiter's HST algorithm can be optimized by using the technique of **explanation reuse**[5] [95]: let v be a newly generated node, if $H(v)$ does not intersect with a previously found explanation, then that explanation can be reused to label v. This is

[5]The authors of [95] called this optimization technique **justification reuse**.

because $H(v)$ represents the set of axioms removed from the ontology, and if none of these removed axioms are present in a particular explanation, then that explanation could still be inferred from the ontology. However, we have already obtained that explanation, therefore there is no need to perform inference again. Explanation reuse helps to reduce the number of calls to SINGLEMINA[6] (Algorithm 8.2).

From the point of view of the implementation, OWLAPI, a famous Java library used to handle OWL ontologies, already provides the class `HSTExplanationGenerator` that implements the hitting set algorithm as described in Algorithm 8.4 and proposed by Kalyanpur in [90]. Moreover it also implements the explanation reuse optimization [95]. This class uses a class that implements function SINGLEMINA. BUNDLE, presented in Chapter 13, exploits `HSTExplanationGenerator` to obtain the set of explanations.

8.2.4 Pinpointing formula

Instead of finding ALL-MINAS$(Q, \mathcal{K})$ for queries, in [100, 101] Baader and Peñaloza proposed the problem of finding a *pinpointing formula* which is a compact representation of the set of all MinAs. To build a pinpointing formula, first we have to associate a unique propositional variable to every axiom E of the KB $\mathcal{K}$, indicated with $var(E)$. Let $var(\mathcal{K})$ be the set of all the propositional variables associated with axioms in $\mathcal{K}$, then the pinpointing formula is a *monotone Boolean formula* built using some or all of the variables in $var(\mathcal{K})$ and the conjunction and disjunction connectives. A valuation ν of a set of variables $var(\mathcal{K})$ is the set of propositional variables that are true, i.e., $\nu \subseteq var(\mathcal{K})$. For a valuation $\nu \subseteq var(\mathcal{K})$, let $\mathcal{K}_\nu := \{E \in \mathcal{K} | var(E) \in \nu\}$.

Definition 8.3 Pinpointing formula
Given a query Q and a KB $\mathcal{K}$, a monotone Boolean formula ϕ over $var(\mathcal{K})$ is called a *pinpointing formula* for Q if, for every valuation $\nu \subseteq var(\mathcal{K})$, $\mathcal{K}_\nu \models Q$ iff ν satisfies ϕ. □

In [101] the authors also discuss the correspondence between the pinpointing formula and explanations for a query Q. Let us call the set of explanations for Q ALL-MINAS$(\mathcal{K}, Q) = \{\mathcal{K}_\nu | \nu$ is a minimal valuation satisfying $\psi\}$. ALL-MINAS$(\mathcal{K}, Q)$ can be obtained by transforming the pinpointing formula into Disjunctive Normal Form (DNF) and removing disjuncts implying other disjuncts. However, it is well-known that this can cause an exponential blowup. The correspondence holds also in the other direction: the formula $\bigvee_{Ex \in \text{ALL-MINAS}(\mathcal{K},Q)} \bigwedge_{E \in Ex} var(E)$ is a pinpointing formula, where Ex is an explanation.

The example below illustrates the difference between ALL-MINAS$(Q, \mathcal{K})$ and the pinpointing formula.

Example 8.2.3 Pinpointing formula

[6]Recall that $\mathcal{SROIQ}$ DLs are `N2ExpTime` complex and every call to SINGLEMINA means that we have to perform inference.

The following KB is inspired by the ontology `people+pets` *[102]:*

$$E_1 = \exists \mathsf{hasAnimal.Pet} \sqsubseteq \mathsf{NatureLover}$$
$$E_2 = \mathsf{fluffy} : \mathsf{Cat}$$
$$E_3 = \mathsf{tom} : \mathsf{Cat}$$
$$E_4 = \mathsf{Cat} \sqsubseteq \mathsf{Pet}$$
$$E_5 = (\mathsf{kevin}, \mathsf{fluffy}) : \mathsf{hasAnimal}$$
$$E_6 = (\mathsf{kevin}, \mathsf{tom}) : \mathsf{hasAnimal}$$

The KB indicates that the individuals that own an animal which is a pet belong to the class NatureLovers *and that* kevin *owns the animals* fluffy *and* tom. fluffy *and* tom *are cats and all the cats are pets. We associated each axiom in the KB with a Boolean variable* E_i*. Let* $Q =$ kevin:NatureLover *be our query, then*

$$\text{All-MinAs(Q,}\mathcal{K}) = \{\{E_2, E_4, E_6, E_1\}, \{E_3, E_5, E_6, E_1\}\}$$

while the pinpointing formula is $((E_2 \wedge E_4) \vee (E_3 \wedge E_5)) \wedge E_6 \wedge E_1$*. It is easy to see that the pinpointing formula is equivalent to* $((E_2 \wedge E_4 \wedge E_6 \wedge E_1) \vee (E_3 \wedge E_5 \wedge E_6 \wedge E_1))$ *that corresponds to* $\text{All-MinAs}(Q, \mathcal{K})$*.*

One interesting feature of the pinpointing formula is that an exponential number of explanations can be represented with a much smaller pinpointing formula.

Example 8.2.4
Given an integer $n \geq 1$*, consider the following KB containing the following axioms for* $1 \leq i \leq n$

$$(C_{1,i})\ B_{i-1} \sqsubseteq P_i \sqcap Q_i \qquad (C_{2,i})\ P_i \sqsubseteq B_i \qquad (C_{3,i})\ Q_i \sqsubseteq B_i$$

The query $Q = B_0 \sqsubseteq B_n$ *has* 2^n *explanations, even if the KB has a size that is linear in* n*. For* $n = 2$ *for example, we have 4 different explanations, namely*

$$\{\{C_{1,1}, C_{2,1}, C_{1,2}, C_{2,2}\}, \{C_{1,1}, C_{3,1}, C_{1,2}, C_{2,2}\}, \{C_{1,1}, C_{2,1}, C_{1,2}, C_{3,2}\}, \{C_{1,1}, C_{3,1}, C_{1,2}, C_{3,2}\}\}$$

The corresponding pinpointing formula is $C_{1,1} \wedge (C_{2,1} \vee C_{3,1}) \wedge C_{1,2} \wedge (C_{2,2} \vee C_{3,2})$ *which is linear in* n*.*

8.2.4.1 The Tableau Algorithm for the Pinpointing Formula

As already said, one of the most common approaches for performing inference in DL is the tableau algorithm. However we have to extend the standard tableau in order to obtain the pinpointing formula.

In particular every assertion $\alpha = n : C$ ($\alpha = (n, m) : R$) with $C \in \mathcal{L}(n)$ ($R \in \mathcal{L}((n, m))$) is associated with a label $lab(a)$ that is a monotone Boolean formula over $var(\mathcal{K})$. In the initial tableau, every assertion $\alpha \in \mathcal{K}$ is labeled with variable $var(\alpha)$, and the assertional negation α_Q (i.e. $\neg Q$) is added with label $\top$.

The tableau is then expanded with expansion rules. A rule is of the form $(B_0, S) \rightarrow \{B_1, ..., B_m\}$ where the B_is are finite sets of assertions and S is a finite set of axioms.

Rules can be divided into two sets: deterministic and non-deterministic. In the first type, $m = 1$ and all the assertions in B_1 are inserted in the tableau to which the rule is applied, while in the second type $m > 1$ meaning that it creates m new tableaux, one for each B_i, and adds to the i-th tableau the assertions in B_i.

In order to explain the conditions that allow the application of a rule we need first some definitions.

Definition 8.4 ***ψ−insertability***
Let A be a set of labeled assertions and ψ a monotone Boolean formula, the assertion α is $\psi-$*insertable* into A if either $\alpha \notin A$, or $\alpha \in A$ but $\psi \not\models lab(\alpha)$. Given a set B of assertions and a set A of labeled assertions, the set of $\psi-$*insertable* elements of B into A is defined as $ins_\psi(B, A) := \{\beta \in B \mid \beta is\ \psi-insertable\ into\ A\}$.

The operation of $\psi-$*insertion* of B into A is the set of labeled assertions $A \uplus_\psi B$ containing assertions in A and those specified in $ins_\psi(B, A)$ opportunely labeled, i.e., the label of assertions in $A \setminus ins_\psi(B, A)$ remain unchanged, assertions in $ins_\psi(B, A) \setminus A$ get label ψ and the remaining b_is get the label $\psi \vee lab(b_i)$. □

We also need the concept of *substitution*. A substitution is a mapping $\rho : V \to D$, where V is a finite set of logical variables and D is a countably infinite set of all the individuals in the KB and all the anonymous individuals created by the application of the rules. Variables are seen as placeholders for individuals in the assertions. For example, an assertion can be $x : C$ or $(x, y) : R$ where C is a concept, R is a role and x and y are variables. Let $x : C$ be an assertion with variable x and $\rho = \{x \to a\}$ a substitution, then $(x : C)\rho$ denotes the assertion obtained by replacing variable x with its $\rho-$*image*, i.e. $(x : C)\rho = a : C$.

Definition 8.5 Rule Applicability
Given a tableau $\mathcal{T}$, a rule $(B_0, S) \to \{B_1, ..., B_m\}$ is applicable with a substitution ρ on the variable occurring in B_0 if $S \subseteq \mathcal{K}$, and $B_0\rho \subseteq A$, where A is the set of assertions of the tableau. □

To explain the motivations behind the definition of *rule applicability* (Definition 8.5), let us consider the following example.

Example 8.2.5
Let us assume A be the current set of labeled assertions $A = \{(a : \exists R.C), ((a, b) : R)\}$. Consider the rule for existential role restrictions:

$$if\ (\{(x : \exists R.C)\}, \{\}) \to \{\{((x, y) : R), (y : C)\}\}$$

The variables x, y are place holders for individuals. To apply the rule to a set of assertions, we must first replace the variables by appropriate individuals. However y occurs only on the right-hand side of the rule, such a variable is called fresh variable. Fresh variables must be replaced by fresh individuals, i.e. new individuals not appearing in A. If we apply the substitution $\rho = \{x \to a, y \to c\}$ that replaces x by a and y by the new anonymous individual c. Since $(x : \exists R.C) \in A$ we can apply the above rule with substitution ρ obtaining the set of assertions $A' = A \cup \{((a, c) : R), (c : C)\}$. Of course, we do not want to apply the same rule to A'. If we do not impose any rule applicability condition, nothing would prevent us from applying the rule to A' with

another new constant, say c', and so on ad infinitum. For this reason, the applicability rule condition needs to check whether the assertions obtained from the right-hand side by substituting the fresh variables with existing individuals are assertions already present in the current set of labeled assertions.

Given a forest of tableaux $\mathcal{F}$ and a tableau $\mathcal{T} \in \mathcal{F}$ representing the set of assertions A to which a rule is applicable with substitution ρ, the application of the rule leads to the new forest $\mathcal{F}' = \mathcal{F} \setminus \mathcal{T} \cup_{i=1}^{n} \mathcal{T}_i^{\psi}$, where each $\mathcal{T}_i^{\psi}$ contains the assertions in $A \uplus_{\psi} B_i\sigma$. The substitution σ extends substitution ρ to map variables to possibly fresh individuals.

After the full expansion of the forest of tableaux, i.e., when no more rules are applicable to any tableau of the forest, the pinpointing formula is built from all the clashes in the tableaux. As mentioned before, a clash is represented by two assertions α and $\neg\alpha$ present in the tableau.

The pinpointing formula is built by first conjoining, for each clash, the labels of the two clashing assertions, by disjoining the formulas for every clash in a tableau and by conjoining the formulas for each tableau.

In order to ensure termination of the algorithm, blocking must be used.

Definition 8.6 Blocking
Given a node N of a tableau, N is blocked iff either N is a new node generated by a rule, it has a predecessor N' which contains the same assertions of N and the labels of these assertions are equal, or its parent is blocked. □

Then, a new definition of applicability must be given.

Definition 8.7 Rule Applicability with Blocking
A rule is applicable if it is so in the sense of Definition 8.5. Moreover, if the rule adds a new node to the tableau, the node N annotated with the assertion to which the rule is applied must be not blocked. □

Theorem 8.3 Correctness (Th. 6.10 [101])
Given a KB $\mathcal{K}$ and a query Q, for every chain of rule applications resulting in a fully expanded forest $\mathcal{F}_n$, the formula built as indicated above is a pinpointing formula for the query Q.

This approach is correct and terminating for the $\mathcal{SHI}$ DL, which extends $\mathcal{ALC}$ with transitive and inverse roles and allows the definition of hierarchies of roles. Number restrictions and nominal concepts cannot be handled by this definition of the tableau algorithm because the definition of rule and rule application must be extended. In fact tableau expansion rules for DL with these constructs may merge some nodes, operation that is not allowed by the approach presented above. The authors of [101] conjecture that the approach can be extended to deal with such constructs but, to the best of our knowledge, this conjecture has not been proved yet.

The expansion rules for the tableau algorithm extended with pinpointing formula are shown in Figure 8.4.

Deterministic rules:
$\rightarrow$ *unfold*: $(\{(x : C)\}, \{(C \sqsubseteq D)\}) \rightarrow \{\{(x : D)\}\}$
if $C \in \mathcal{L}(x)$, and $(C \sqsubseteq D) \in K$, **then** $\psi := lab(x : C) \wedge var((C \sqsubseteq D))$
  if D $\notin \mathcal{L}(x)$, **then** $add(D, x)$, $lab(x : D) := \psi$
  else if $\psi \not\models lab(x : D)$, **then** $lab(x : D) := lab(x : D) \vee \psi$
$\rightarrow$ *CE*: $(\{\}, \{(C \sqsubseteq D)\}) \rightarrow \{\{(x : (\neg C \sqcup D))\}\}$
if $(C \sqsubseteq D) \in K$, with C not atomic, **then**
  if $(\neg C \sqcup D) \notin \mathcal{L}(x)$, **then** $add((\neg C \sqcup D), x)$, $lab(x : (\neg C \sqcup D)) := var((C \sqsubseteq D))$
$\rightarrow \sqcap$: $(\{(x : (C_1 \sqcap C_2))\}, \{\}) \rightarrow \{\{(x : C_1), (x : C_2)\}\}$
if $(C_1 \sqcap C_2) \in \mathcal{L}(x)$, **then** $\psi := lab(x : (C_1 \sqcap C_2))$
  if $C_i \notin \mathcal{L}(x)$, **then** $add(C_i, x)$, $lab(x : C_i) := \psi$
  else if $\psi \not\models lab(x : C_i)$, **then** $lab(x : C_i) := lab(x : C_i) \vee \psi$
$\rightarrow \exists$: $(\{(x : \exists R.C)\}, \{\}) \rightarrow \{\{((x, y) : R), (y : C)\}\}$
if $\exists S.C \in \mathcal{L}(x)$, x not blocked, **then**
  if x has no S-neighbour y with $C \in \mathcal{L}(y)$, **then** create new node y, $add(S, (x, y))$,
    $add(C, y)$, $lab(y : C) := lab(x : (\exists S.C))$, $lab((x, y) : S) := lab(x : (\exists S.C))$
$\rightarrow \forall$: $(\{(x : \forall S.C)\}, \{((x, y) : S)\}) \rightarrow \{\{(y : C)\}\}$
if $\forall S.C \in \mathcal{L}(x)$ and there is an S-neighbour y of x, **then**
  $\psi := lab((\forall S.C), x) \wedge lab((x, y) : S)$
  if $C \notin \mathcal{L}(y)$, **then** $add(C, y)$, $lab(y : C) := \psi$
  else if $\psi \not\models lab(y : C)$, **then** $lab(y : C) := lab(y : C) \vee \psi$
$\rightarrow \forall^+$: $(\{(x : \forall S.C)\}, \{((x, y) : R), (R \sqsubseteq S)\}) \rightarrow \{\{(y : \forall R.C)\}\}$
if $\forall(S.C) \in \mathcal{L}(x)$, there is an R-neighbour y of a, $Trans(R)$ and $R \sqsubseteq S$, **then**
  $\psi := lab(x : (\forall S.C)) \wedge lab((x, y) : R) \wedge var(R \sqsubseteq S)$
  if $\forall R.C \notin \mathcal{L}(y)$, **then** $add((\forall R.C), y)$, $lab(y : (\forall R.C)) := \psi$
  else if $\psi \not\models lab(y : (\forall R.C))$, **then** $lab(y : (\forall R.C)) := lab(y : (\forall R.C)) \vee \psi$
Non-deterministic rules:
$\rightarrow \sqcup$: $(\{(x : (C_1 \sqcup C_2))\}, \{\}) \rightarrow \{\{(x : C_1)\}, \{(x : C_2)\}\}$
if $(C_1 \sqcup C_2) \in \mathcal{L}(x)$, **then**
  if $\{C_1, C_2\} \cap \mathcal{L}(a) = \emptyset$, **then** generate graphs $\mathcal{T}_i := \mathcal{T}$ for each $i \in \{1, 2\}$,
    $add(C_i, x)$ in $\mathcal{T}_i$ for each $i \in \{1, 2\}$, $lab(x : C_i) := lab(x : (C_1 \sqcup C_2))$

Figure 8.4: Tableau expansion rules for building a pinpointing formula. Function $add(X, Y)$ adds X to $\mathcal{L}(Y)$.

8.3 Conclusions

All standard reasoning problems can be reduced to a consistency check of the knowledge base. In this chapter we presented the tableau algorithm which tries to construct models of a given knowledge base by applying tableau expansion rules. If it succeeds, the knowledge base is consistent and hence satisfiable, otherwise, if it fails, the knowledge base is inconsistent and hence unsatisfiable. We also showed a method for finding explanations based on the hitting set tree algorithm, and one to obtain the pinpointing formula. Both leverage the tableau algorithm, the former can be used for $\mathcal{SROIQ}(\mathbf{D})$ knowledge bases whereas the latter is limited to $\mathcal{SHI}$ knowledge bases.

In the next chapter we discuss a semantics for probabilistic description logics that tries to combine DLs with probability theory.

Chapter 9

Probabilistic Description Logics

This chapter introduces DISPONTE, a semantics for Probabilistic Description Logics. After a brief introduction to the topics in Section 9.1, Section 9.2 presents DISPONTE. Section 9.3 discusses related works and Section 9.4 concludes the chapter.

9.1 Introduction

The goal of this part of the thesis is to introduce probabilistic logical formalisms for representing uncertainty. Chapter 5 described the *distribution semantics* and in Chapter 6 we saw that a probabilistic logic program following the distribution semantics [1] defines a probability distribution over normal logic programs called *worlds*. The probability of a query is obtained by marginalizing the joint distribution of the query and the worlds.

DISPONTE [2], for "DIstribution Semantics for Probabilistic ONTologiEs", borrows the work done in Probabilistic Logic Programming (PLP) and adapts the *distribution semantics* to description logics.

9.2 The Distribution Semantics for Description Logics: DISPONTE

The basic idea of DISPONTE is to annotate axioms with a probability and each axiom is assumed to be independent of the others. Here we show its syntax and semantics of DISPONTE.

9.2.1 Syntax

In DISPONTE, a *probabilistic knowledge base* $\mathcal{K}$ is a set of certain axioms or probabilistic axioms. Certain *axioms* take the form of regular DL axioms. *Probabilistic axioms* take the form

$$p :: E$$

where $p \in [0, 1]$ and E is a DL axiom.

The probabilistic knowledge that can be expressed with the DISPONTE semantics is **epistemic** by nature, it represents degrees of belief in the axioms rather than statistical information. For example, a probabilistic concept assertion axiom

$$p :: a : C$$

means that we have degree of belief p in $a : C$. The statement that Tweety flies with probability 0.9 can be expressed as

$$0.9 :: \mathsf{tweety} : \mathsf{Flies}$$

A probabilistic concept subsumption axiom of the form

$$p :: C \sqsubseteq D \tag{9.1}$$

represent the fact that we believe that the axiom $C \sqsubseteq D$ is true with probability p. For example

$$0.9 :: \mathsf{Bird} \sqsubseteq \mathsf{Flies}$$

means that birds fly with a 90% probability. This is different from **statistical** probability [103] that express the degree of overlap of C and D. Axiom (9.1) does not mean that a fraction p of individuals from C belongs to D.

9.2.2 Semantics

DISPONTE follows the approach of the distribution semantics for probabilistic logic programs. The idea is to associate independent Boolean random variables to the DL axioms. The set of axioms that have the random variable assigned to 1 constitutes a *world*.

The definitions of *atomic choice*, *composite choice*, *selection*, etc. for Probabilistic Description Logics (PDLs) that follow DISPONTE are slightly different from those given for PLP (see Chapter 6). We give here some *redefinitions* used for PDLs that follow DISPONTE.

Definition 9.1 Atomic choice
An *atomic choice* is a couple (E_i, k) where E_i is the ith probabilistic axiom and $k \in \{0, 1\}$. The variable k indicates whether E_i is chosen to be included in a world ($k = 1$) or not ($k = 0$). □

Note that the definition of atomic choice for PDLs given in Definition 9.1 is very similar to the definition of atomic choice for LPADs in Definition 6.1. The only difference is that atomic choices for PDLs do not have a substitution, as we include or exclude axioms as a whole instead of their instantiations.

Definition 9.2 Composite choice
A *composite choice* κ is a consistent set of atomic choices, i.e., $(E_i, k) \in \kappa$, $(E_i, m) \in \kappa$ implies $k = m$ (only one decision is taken for each formula). The probability of composite choice κ is

$$P(\kappa) = \prod_{(E_i,1)\in\kappa} p_i \prod_{(E_i,0)\in\kappa} (1 - p_i)$$

where p_i is the probability associated with axiom E_i, because the random variables associated with axioms are independent. □

Definition 9.3 Selection
A *selection* σ is a total composite choice, i.e., it contains an atomic choice (E_i, k) for every probabilistic axiom of the theory. A *selection* σ identifies a theory w_σ called a *world*: $w_\sigma = \mathcal{C} \cup \{E_i | (E_i, 1) \in \sigma\}$, where $\mathcal{C}$ is the set of certain axioms. □

The probability of a world w_σ is analogous to Equation (6.3):

$$P(w_\sigma) = P(\sigma) = \prod_{(E_i,1)\in\sigma} p_i \prod_{(E_i,0)\in\sigma} (1 - p_i) \tag{9.2}$$

As for PLP, $P(w_\sigma)$ is a probability distribution over worlds. Let us indicate with $\mathcal{S}$ and $\mathcal{W}$ the set of all selection and the set of all worlds, respectively. The probability of Q is (equal to Equation (6.4)):

$$P(Q) = \sum_{w\in\mathcal{W}:w\models Q} P(w)$$

Example 9.2.1
Consider another example inspired by the `people+pets` *ontology proposed in [102]*

$$\begin{aligned} E_1 &= 0.5 :: \exists \mathsf{hasAnimal.Pet} \sqsubseteq \mathsf{NatureLover} \\ E_2 &= 0.6 :: \mathsf{Cat} \sqsubseteq \mathsf{Pet} \\ &\quad (\mathsf{kevin}, \mathsf{fluffy}) : \mathsf{hasAnimal} \\ &\quad (\mathsf{kevin}, \mathsf{tom}) : \mathsf{hasAnimal} \\ &\quad \mathsf{fluffy} : \mathsf{Cat} \\ &\quad \mathsf{tom} : \mathsf{Cat} \end{aligned}$$

The KB indicates that the individuals that own an animal which is a pet are nature lovers with a 50% probability and that Kevin has the animals fluffy and tom. Fluffy and tom are cats and cats are pets with probability 60%. The KB has four possible worlds:

$$\begin{aligned} \sigma_1 &= \{(E_1, 1), (E_2, 1)\} \\ \sigma_2 &= \{(E_1, 1), (E_2, 0)\} \\ \sigma_3 &= \{(E_2, 0), (E_2, 1)\} \\ \sigma_4 &= \{(E_2, 0), (E_2, 0)\} \end{aligned}$$

The query axiom $Q = \mathsf{kevin} : \mathsf{NatureLover}$ *is true only in one of them:* σ_1. *The probability of the query is*

$$P(Q) = 0.5 \cdot 0.6 = 0.3$$

.

Example 9.2.2
Let us consider the same knowledge base of Example 9.2.1 but with different probabilistic values:

$$\begin{aligned} & \exists \mathsf{hasAnimal}.\mathsf{Pet} \sqsubseteq \mathsf{NatureLover} \\ E_1 = \; & 0.6 :: \mathsf{Cat} \sqsubseteq \mathsf{Pet} \\ & (\mathsf{kevin}, \mathsf{fluffy}) : \mathsf{hasAnimal} \\ & (\mathsf{kevin}, \mathsf{tom}) : \mathsf{hasAnimal} \\ E_2 = \; & 0.4 :: \mathsf{fluffy} : \mathsf{Cat} \\ E_3 = \; & 0.3 :: \mathsf{tom} : \mathsf{Cat} \end{aligned}$$

This KB indicates that the individuals that own an animal which is a pet are surely nature lovers and we know for sure that kevin *has the animals* fluffy *and* tom*, but we are not sure that* fluffy *and* tom *are cats and that cats are pets, thus we believe in this information is probabilistic.*

This KB has eight worlds[1] *and the query axiom* $Q = \mathsf{kevin} : \mathsf{NatureLover}$ *is true in three of them, corresponding to the following selections:*

$$\begin{aligned} \sigma_1 &= \{(E_1, 1), (E_2, 1), (E_3, 0)\} \\ \sigma_2 &= \{(E_1, 1), (E_2, 0), (E_3, 1)\} \\ \sigma_3 &= \{(E_1, 1), (E_2, 1), (E_3, 1)\} \end{aligned}$$

so the probability is

$$P(Q) = 0.4 \cdot 0.7 \cdot 0.6 + 0.6 \cdot 0.3 \cdot 0.6 + 0.4 \cdot 0.3 \cdot 0.6 = 0.348.$$

In Section 6.2 we said that often is not possible to find all the worlds where the query is true, so an approach for performing inference is to find the *explanations* for the query and then compute the probability of the query from them.

Definition 9.4 Explanation (for PDLs)
A composite choice κ *identifies* a set of worlds $\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_\mathcal{K}, \sigma \supseteq \kappa\}$, if Q is entailed by every world of ω_κ, then κ is called *explanation*. □

A set of composite choices K is *covering* Q if every world $w_\sigma \in \mathcal{W}$ in which Q is entailed is such that $w_\sigma \in \bigcup_{\kappa \in K} \omega_\kappa$. In other words a covering set K identifies all the worlds in which Q succeeds.

Two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent. For Example $\kappa_1 = \{(E_i, 1)\}$ and $\kappa_2 = \{(E_i, 0)\}$ are incompatible. A set K of composite choices is *pairwise incompatible* if for all $\kappa_1 \in K$, $\kappa_2 \in K$, $\kappa_1 \neq \kappa_2$ implies that κ_1 and κ_2 are incompatible. The *probability of a pairwise incompatible set of composite choices* K is defined in Equation (6.5).

Given a query Q and a covering set of pairwise incompatible composite choices K, the probability of Q is defined in Equation (6.7).

[1]The number of words is obtained with 2^n where n is the number of probabilistic axioms.

Example 9.2.3
Consider the DL KB in Example 9.2.2 and the same query $Q = \mathsf{kevin} : \mathsf{NatureLover}$. *The set of covering explanations is*

$$K = \{\kappa_1, \kappa_2\}$$
$$\kappa_1 = \{(E_1, 1), (E_2, 1)\}$$
$$\kappa_2 = \{(E_1, 1), (E_3, 1)\}$$

These explanations are not pairwise incompatible, therefore we cannot compute the probability with Equation (6.7). In fact

$$P(\kappa_1) + P(\kappa_2) = 0.6 \cdot 0.4 + 0.6 \cdot 0.3 = 0.42 \neq 0.348 = P(Q)$$

where $P(Q)$ *was computed in Example 9.2.2. If we knew to obtain the following mutually exclusive explanations*

$$K' = \{\kappa'_1, \kappa'_2\}$$
$$\kappa'_1 = \{(E_1, 1), (E_2, 1), (E_3, 0)\}$$
$$\kappa'_2 = \{(E_1, 1), (E_3, 1)\}$$

we would have

$$P(K') = P(\kappa'_1) + P(\kappa'_2) = 0.6 \cdot 0.4 \cdot 0.7 + 0.6 \cdot 0.3 = 0.348 = P(Q)$$

DISPONTE, like the distribution semantics, defines a distribution over worlds. Hence, it is worth noticing that, if the regular DL KB obtained by removing the probabilistic annotations is inconsistent, then there will also be worlds that are inconsistent. An inconsistent DISPONTE KB should not be used to derive consequences, just as a regular inconsistent DL KB should not.

In order to specify requirements for managing uncertain information in the World Wide Web, W3C, in 2007, founded the Uncertainty Reasoning for the World Wide Web Incubator Group (URW3-XG). The final report produced by this group in 2008 [104] discusses the challenges of reasoning with uncertain information on the World Wide Web and presents several use cases illustrating conditions under which uncertainty reasoning is important:

- combining knowledge from multiple, untrusted sources;
- recommending items or services to users in the presence of uncertain information on the requirements;
- using services in the presence of uncertain information on the service descriptions;
- extracting and annotating information from the web;
- automatically performing tasks for users such as making an appointment, and handling health-care and life sciences information and knowledge.

DISPONTE is a candidate formalism for tackling these problems since it introduces probability in form of simple axiom annotations for very expressive Description Logics such as $\mathcal{SROIQ}(\mathbf{D})$ that, as mentioned previously in Chapter 7, is semantically equivalent to OWL 2 DL. Moreover DISPONTE can handle information coming from different untrusted sources, as shown in the following example.

Example 9.2.4 [92]
Consider a KB similar to the one of Example 9.2.1. Suppose the user has the certain knowledge

$$\exists \mathsf{hasAnimal.Pet} \sqsubseteq \mathsf{NatureLover}$$
$$(\mathsf{kevin,fluffy}) : \mathsf{hasAnimal}$$
$$\mathsf{Cat} \sqsubseteq \mathsf{Pet}$$

In this example there are two sources of information with different reliability. The two sources represent independent evidence on fluffy *being a cat. One source state that* fluffy *is a cat with a degree of belief of 0.4, whereas on the other source we have a degree of belief of 0.3. The user can add the following statements to his KB and reason on the new knowledge :*

$$E_1 = 0.4 :: fluffy : Cat$$
$$E_2 = 0.3 :: fluffy : Cat$$

The query axiom $Q = \mathsf{kevin} : \mathsf{NatureLover}$ *is true in three out of the four worlds, those corresponding to the selections:*

$$\{(E_1, 1), (E_2, 1)\}$$
$$\{(E_1, 1), (E_2, 0)\}$$
$$\{(E_1, 0), (E_2, 1)\}$$

So

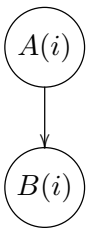
$$P(Q) = 0.4 \cdot 0.3 + 0.4 \cdot 0.7 + 0.6 \cdot 0.3 = 0.58$$

This is reasonable if the two sources can be considered as independent. In fact, If we associate with E_1 *and* E_2 *two Boolean random variables* X_1 *and* X_2 *with probabilities respectively 0.4 and 0.3, we obtain:*

$$\begin{aligned} P(Q) &= P(X_1 \vee X_2) \\ &= P(X_1) + P(X_2) - P(X_1 \wedge X_2) \\ &= P(X_1) + P(X_2) - P(X_1)P(X_2) \\ &= 0.4 + 0.3 - 0.4 \cdot 0.3 = 0.58 \end{aligned}$$

9.2.3 Assumption of Independence

The assumption of the independence of the axioms may seem restrictive. However, any probabilistic relationship between assertions that can be represented with a Bayesian network can be modeled in this way. For example, suppose you want to model a general



Prob. Table 1:

$A(i)$	$P(A(i))$
0	$1-p_1$
1	p_1

Prob. Table 2:

$B(i)$ \ $A(i)$	0	1
0	$1-p_2$	$1-p_3$
1	p_2	p_3

Figure 9.1: Bayesian Network representing the dependency between $A(i)$ and $B(i)$.

dependency between the assertions $A(i)$ and $B(i)$ relating classes A and B to individual i. This dependency can be represented with the Bayesian network of Figure 9.1.

The joint probability distribution $P(A(i), B(i))$ over the two Boolean random variables $A(i)$ and $B(i)$ is

$$\begin{aligned} P(0,0) &= (1-p_1)\cdot(1-p_2) \\ P(0,1) &= (1-p_1)\cdot(p_2) \\ P(1,0) &= p_1\cdot(1-p_3) \\ P(1,1) &= p_1\cdot p_3 \end{aligned}$$

This dependence can be modeled with the following DISPONTE KB:

$$p_1 :: i : A \tag{9.3}$$
$$p_2 :: \neg A \sqsubseteq B \tag{9.4}$$
$$p_3 :: A \sqsubseteq B \tag{9.5}$$

We can associate the Boolean random variables X_1 with (9.3), X_2 with (9.4) and X_3 with (9.5), where X_1, X_2 and X_3 are mutually independent. These three random variables generate 8 worlds. $\neg A(i) \wedge \neg B(i)$ is true in the worlds

$$w_1 = \{\},\ w_2 = \{(9.5)\}$$

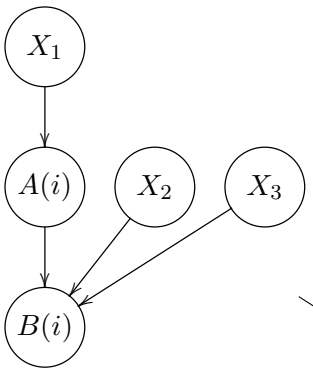
Let us call P' the probability distribution defined by the above KB. Then the probabilities of w_1 and w_2 are

$$\begin{aligned} P'(w_1) &= (1-p_1)\cdot(1-p_2)\cdot(1-p_3) \\ P'(w_2) &= (1-p_1)\cdot(1-p_2)\cdot p_3 \end{aligned}$$

so $P'(\neg A(i), \neg B(i)) = (1-p_1)\cdot(1-p_2)\cdot(1-p_3) + (1-p_1)\cdot(1-p_2)\cdot p_3 = P(0,0)$. We can prove similarly that the distributions P and P' coincide for all joint states of $A(i)$ and $B(i)$.

Modeling the dependency between $A(i)$ and $B(i)$ with the KB above is equivalent to represent the Bayesian network of Figure 9.1 with the Bayesian network of Figure 9.2.

It can be easily checked that the distributions P and P'' of the two networks agree on the variables $A(i)$ and $B(i)$, i.e., that $P(A(i), B(i)) = P''(A(i), B(i))$ for any value of $A(i)$ and $B(i)$. From Figure 9.2 is also clear that X_1, X_2 and X_3 are mutually unconditionally independent, thus showing that it is possible to represent any dependence with independent random variables. Therefore we can model general dependencies among assertions with DISPONTE.



Prob. Table 3: $P''(X_1)$

X_1	
0	$1 - p_1$
1	p_1

Prob. Table 4: $P''(X_2)$

X_2	
0	$1 - p_2$
1	p_2

Prob. Table 5: $P''(A(i)|X_1)$

$A(i)$ \ X_1	0	1
0	1	0
1	0	1

Prob. Table 6: $P''(X_3)$

X_3	
0	$1 - p_3$
1	p_3

Prob. Table 7: $P''(B(i)|X_1, X_2, X_3)$

$B(i)$ \ X_1, X_2, X_3	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
0	1	1	0	0	1	0	1	0
1	0	0	1	1	0	1	0	1

Figure 9.2: Bayesian Network modeling the distribution over $A(i)$, $B(i)$, X_1, X_2, X_3.

9.3 Related Work

Bacchus [105] and Halpern [103] proposed a classification between different types of probabilistic first-order logics. They defined two types of probability:

Statistical This type of probability[2] allows to express *statistical* information. It puts probability on entities of the domain and permits the definition of statement such as "The probability that a randomly chosen bird will fly is 0.9". It means that 90% of the birds in a population can fly.

Epistemic This type of probability[3] allows to express *epistemic* information which defines a *degree of belief*. It puts probability on possible world such that we can assert statement such as "The probability that Tweety flies is .9". This sentence mean that in 90% of possible worlds we have that Tweety can fly.

Assertional axiom can only be epistemic, whereas the intensional axioms (i.e. axioms in TBox and RBox) can be both statistical and epistemic.

DISPONTE allows to express only epistemic information since the probability associated with an axiom represent the degree of belief that the whole axiom is true.

[2]In [103] the author used the term *Type 1*.
[3]In [103] the author used the term *Type 2*.

During the years, several frameworks emerged in order to conjugate DLs with probability theory. Here we provide a summary of the possible PDLs. A more detailed and complete overview about probabilistic DLs can be found in [3], in Chapter 15 of [91] and in Chapter 13 of [92].

Halpern in [103] proposed a probabilistic extension of OWL for combining the two types of probabilities. In this extension is possible to define statements such as "The probability that Tweety flies is greater than the probability that a randomly chosen bird flies".

Prob-$\mathcal{ALC}$ [106] derives directly from Halpern's work and considers only epistemic statements. It follows a possible world semantics and allows the definition of concept expressions of the form $P_{\geq n}C$ which express the set of individuals that belong to C with probability greater than n, and $\exists P_{\geq n}R.C$ which models set of individuals a connected to at least another individual b of C by role R such that the probability of $R(a,b)$ is greater than n. Prob-$\mathcal{ALC}$ allows also expressions of the form $P_{\geq n}C(a)$ and $P_{\geq n}R(a,b)$ directly expressing degrees of belief, as well as $P_{\geq n}\mathcal{A}$ where $\mathcal{A}$ is an ABox. Prob-$\mathcal{ALC}$ is complementary to DISPONTE $\mathcal{ALC}$ as it allows new concept and assertion expressions whereas DISPONTE allows probabilistic axioms. However DISPONTE is not limited to $\mathcal{ALC}$, but it can be applied to the highly expressive $\mathcal{SROIQ}(\mathbf{D})$ language.

Heinsohn [107] extended the DL $\mathcal{ALC}$ in order to allow the definition of statistical information of the form $P(C|D) = [p,q]$ called *probabilistic terminological axioms*, where C, D are concept descriptions and $0 \leq p \leq q \leq 1$ are real numbers. It states that the conditional probability for an object belonging to D of belonging also to C is in the interval $[p,q]$. The formal semantics of the extended language is defined in terms of probability measures on the set of all concept descriptions. A finite interpretation $\mathcal{I}$ satisfies $P(C|D) = [p,q]$ iff

$$\frac{|(C \sqcap D)^{\mathcal{I}}|}{|D^{\mathcal{I}}|} = [p,q]$$

A knowledge base $\mathcal{K}$ consists of probabilistic terminological axioms. Given $\mathcal{K}$, the main inference task is to find an interval $[p,q]$, with p maximal and q minimal. Heinsohn introduced local inference rules for deriving bounds but its approach is not complete, hence the rules are not sufficient to derive optimal bounds. Furthermore, Heinsohn semantics does not allow probabilistic assertional axioms. This limit was overcome by Jaeger in [108]. Unlike DISPONTE, both the approaches in [107] and [108] do not allow epistemic terminological statements.

A different approach to the combination of DLs with probability is taken in [109, 110], which introduces PR-OWL, a probabilistic extension for OWL. In order to represent uncertainty in ontologies it allows the use of the first-order probabilistic logic MEBN [111]. DISPONTE differs from [109, 110] because it does not resort to a full-fledged first-order probabilistic language, allowing the reuse of inference technology from DLs.

In [112, 113, 114] the authors presented P-$\mathcal{SHIQ}(\mathbf{D})$. P-$\mathcal{SHIQ}(\mathbf{D})$ allows both terminological and assertional probabilistic axioms. Terminological probabilistic knowledge is expressed using *conditional constraints* of the form $(D|C)[l,u]$ as in [107] and of the form $(\exists R.\{a\}|C)[l,u]$ that states that an arbitrary instance of a concept C is R-related to the individual a with probability in the interval $[l,u]$. Assertional probabilistic knowledge is expressed using constraints of the form $(C|\{a\})[l,u]$ and

$(\exists R.\{b\}|\{a\})[l,u]$, which represent respectively that the individual a belongs to C and a is R-related to b with a probability in the interval $[l,u]$. As in [108], the terminological knowledge is interpreted statistically while the assertional knowledge is interpreted in an epistemic way.

A different approach is given by the combination between DLs and logic programs. In [115], ontologies are integrated with rules and a tightly coupled approach to (probabilistic) disjunctive description logic programs is used. Under this semantics, a description logic program as a pair (L,P), where L is a DL KB and P is a disjunctive logic program which contains rules on concepts and roles of L.

Other approaches try to convert probabilistic description logics into graphical models. In [116] the authors proposed a probabilistic extension of OWL that can be translated into Bayesian networks. The semantics defines a probability distribution $P(a)$ over individuals and assigns a probability to a class C as $P(C) = \sum_{a \in C} P(a)$. DISPONTE differs from [116] because it specifies a distribution over worlds rather than individuals.

Koeller et al. [117] presented a probabilistic description logic based on Bayesian networks that deals with statistical terminological knowledge. They specify a unique probability distribution on the set of all concept descriptions.

In [118] the authors presented another extension of $\mathcal{ALC}$ named CR$\mathcal{ALC}$. It adopts a semantics based on interpretations and, differently form DISPONTE, allows the expression of both statistical and epistemic probability types. Statistical axioms are of the form $P(C|D) = p$ which means that for any element of the domain, the probability that an individual is in C given that it is in D is p, and of the form $P(R) = p$, which means that for each pair of elements of the domain, the probability that they are linked by the role R is p. A CR$\mathcal{ALC}$ KB $\mathcal{K}$ can be represented as a directed acyclic graph.

Other approaches are the ones proposed in [119] and [120]. The latter combines *Datalog*± with Markov networks, the former, instead, combines *DL-Lite* with Bayesian networks. In both cases, an ontology is composed of a set of annotated axioms and a graphical model and the annotations are sets of assignments of random variables from the graphical model. The semantics is assigned by considering the possible worlds of the graphical model and by stating that an axiom holds in a possible world if the assignments in its annotation hold. The probability of a query is then the sum of the probabilities of the possible worlds where the query holds.

9.4 Conclusions

In this chapter we illustrated a semantics for probabilistic description logics named DISPONTE, which allows to express epistemic probabilistic axioms.

There are different approaches to represent probabilistic axioms in description logics. DISPONTE is, in our opinion, an interesting semantics because it allows to define epistemic probabilities on both assertional and terminological axioms. In addition, It allows the reuse of inference technologies already developed for Description Logics.

This chapter concludes the part of the thesis concerning the techniques of knowledge representation by probabilistic logics. The next part introduces algorithms and systems to make inferences on this types of logics.

Part III

Inference in Probabilistic Logics

Chapter 10

Decision Diagrams

At this point the reader may be eager to know how to perform inference on probabilistic logics. However, before discussing inference, we need to introduce Decision Diagrams. After a brief introduction (Section 10.1), we present Multivalued Decision Diagrams (Section 10.2) and Binary Decision Diagrams (Section 10.3), finally in Section 10.4 we conclude the chapter.

10.1 Introduction

Decision Diagrams are graphical structures widely used in many areas of computer science, such as software and hardware verification. They are used for representing and manipulating propositional logic formulas. They also find application in probabilistic inference systems, in fact they are exploited by ProbLog [30] and PITA [121, 122].

10.2 Multivalued Decision Diagrams

Multivalued Decision Diagrams (MDDs) were introduced by Thayse in [123]. They are rooted directed acyclic graphs which represent a *Boolean-valued function* $f(\mathbf{X})$ having $\mathbb{B} = \{0, 1\}$ as range and $\mathbf{X}$, a set of multivalued variables, as domain.

An MDD has the following characteristics

- There is one level for each multivalued variable.
- Each node is associated with a multivalued variable.
- Each node has one outgoing arc for every possible value of the multivalued variable.
- Two leaves, i.e. terminal nodes, that store either 0 (false) or 1 (true).

If we have the values of all the variables $\mathbf{X}$, the value of $f(\mathbf{X})$ can be computed by traversing the graph starting from the root until a leaf is reached. The value stored in the reached leaf is the value of $f(\mathbf{X})$.

An MDD can be built by combining multiple MDDs exploiting Boolean operators. Moreover, in order to reduce the number of node and have a more compact graphical

representation, it is possible to apply simplification operations like *deletion* and *merging* that *reduce* the original MDD. Deletion is performed when all arcs from a node point to the same node, Merging, instead, is performed when the diagram contains two identical sub-diagrams.

Multivalued Decision Diagrams are really useful because they perform a generalization of Shannon's expansion of the Boolean-valued formula $f(\mathbf{X})$. Let X_i be the variable associated with the root level of a (sub-)MDD, the formula $f(\mathbf{X})$ is expanded as follows

$$f(\mathbf{X}) = (X_i = 1) \wedge f^{X_i=1}(\mathbf{X}) \vee \cdots \vee (X_i = n) \wedge f^{X_i=n}(\mathbf{X})$$

where $f^{X_i=k}(\mathbf{X})$ is the function associated with the k-child of the root node X_i and it is equivalent to $f(\mathbf{X})$ with X_i set to k, i.e. $f^{X_i=k}(\mathbf{X}) = f(\dots, X_{i-1}, X_i = k, X_{i+1}, \dots)$. The expansion can be applied recursively to the functions $f^{X_i=k}(\mathbf{X})$. The disjuncts are now pairwise incompatible due to the presence of $X_i = k$. The paths in an MDD are split on the basis of a multivalued variable and the branches are mutually disjoint.

Unfortunately there is a lack of libraries that can manipulate MDDs, most libraries are limited to work on BDDs. However it is possible to covert an MDD into a BDD and thus use packages for BDD manipulation.

10.3 Binary Decision Diagrams

Binary decision Diagrams were introduced by Akers [124] and Bryant [125, 126]. A BDD is a rooted directed acyclic graph that can represent any Boolean formula $f : \mathbb{B}^n \to \mathbb{B}$, with the following characteristics

- There is one level for each Boolean variable.
- Each node is labeled with a Boolean variable. We denote with $var(n)$ to indicate the variable name associated with node n.
- Each node has two possible children. One *high child* named $child_1(n)$ when $var(n)$ has value 1, and one *low child*, $child_0(n)$, when $var(n)$ has value 0.
- Two leaves, i.e. terminal nodes, that store either 0 (false) or 1 (true).

Boolean functions are a special case of Boolean-valued functions where all the Xs in the domain are Boolean variables. In fact BDDs can be seen as a special case of MDDs in which every variable in the domain can have only two values (0 and 1). As for MDDs, to compute the value of $f(\mathbf{X})$, given all values of $\mathbf{X}$, the graph must be traversed from the root and the returned value is the value associated with the reached leaf.

In literature, in most cases, the term BDD refers to Reduced Ordered Binary Decision Diagrams (ROBDD)[1].

Definition 10.1 Ordered BDD: OBDD
Given a total order $X_1 \prec X_2 \prec \cdots \prec X_n$, A BDD is *ordered* (OBDD) if the variables are encountered through all paths respecting the given total order. □

[1] In the next chapters we also adopt this convention.

Definition 10.2 Reduced BDD: RBDD
A BDD is *reduced* (RBDD) we have the following conditions

- (uniqueness) There are no two distinct nodes u and v such that $var(u) = var(v)$, $child_0(u) = child_0(v)$ and $child_1(u) = child_1(v)$, in other words there are no two distinct nodes that are associated with the same variable and that have the same low child and the same high child.
- (no-redundancy) There is no variable node n such that the low child and the high child are the same node, i.e. $child_1(n) = child_0(n)$.

□

The order of variables affects the size of the (RO)BDD. In order to obtain a compact BDD representable in memory, state-of-the-art BDD packages employ heuristics and optimization techniques to reorder the variables; finding an optimal order is an NP-complete problem [127]. Some of the libraries for BDD handling are:

- CUDD [128], a BDD package written in C. The main characteristic of this library is that it represent the BDDs without the 0-leaf and uses *complement arcs* (see [128]).
- CAL [129], a BDD package written in C.
- BuDDy [130], a BDD package written in C.
- JDD [131], a pure Java implementation for BDD manipulation based on the C library BuDDy.
- JavaBDD [132], it includes a pure Java implementation. Moreover, it can also be used as an interface for the JDD library, or, by exploiting JNI, for the three aforementioned BDD libraries written in C: BuDDy, CAL and CUDD.

BDDs like MDDs perform Shannon's expansion of the Boolean function $f(\mathbf{X})$. Let X_i be the variable associated with the root level of a (sub-)BDD, the function $f(\mathbf{X})$ is expanded as follows

$$f(\mathbf{X}) = X_i \wedge f^{X_i}(\mathbf{X}) \vee \neg X_i \wedge f^{\neg X_i}(\mathbf{X})$$

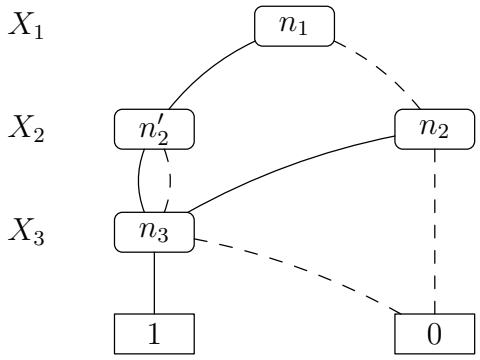
where $f_i^X(\mathbf{X})$ ($f^{\neg X_i}(\mathbf{X})$) is the function obtained by $f(\mathbf{X})$ by setting X_i to 1 (0), i.e. $f^{X_i}(\mathbf{X}) = f(\dots, X_{i-1}, 1, X_{i+1}, \dots)$ (i.e. $f^{\neg X_i}(\mathbf{X}) = f(\dots, X_{i-1}, 0, X_{i+1}, \dots)$). Now the two disjuncts $X_i \wedge f^{X_i}(\mathbf{X})$ and $\neg X_i \wedge f^{\neg X_i}(\mathbf{X})$ are mutually exclusive. The expansion can be applied recursively to the functions $f^{X_i}(\mathbf{X})$ and $f^{\neg X_i}(\mathbf{X})$.

Example 10.3.1
Consider the following ***Disjunctive Normal Form*** *(DNF) Boolean formula*

$$f(\mathbf{X}) = X_1 \wedge X_3 \vee X_2 \wedge X_3$$

and a total order of the variables $X_1 \prec X_2 \prec X_3$*. The disjuncts of* $f(\mathbf{X})$ *are not mutually exclusive, for instance, the model* $\{X_1 = 1, X_2 = 1, X_3 = 1\}$ *makes both the disjunct true.*

The corresponding OBDD is



This is equivalent to the following Shannon expansions of $f(\mathbf{X})$*. For* X_1

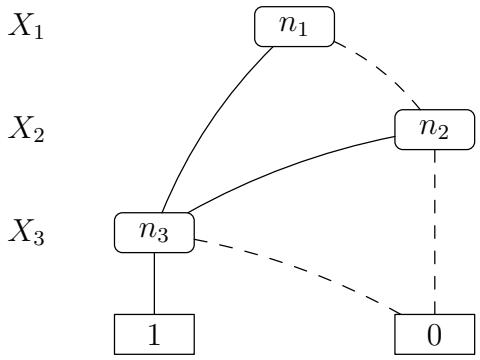
$$\begin{aligned} f(\mathbf{X}) &= X_1 \wedge f^{X_1}(\mathbf{X}) \vee \neg X_1 \wedge f^{\neg X_1}(\mathbf{X}) \\ &= X_1 \wedge X_3 \vee X_1 \wedge X_2 \wedge X_3 \vee \neg X_1 \wedge X_2 \wedge X_3 \end{aligned}$$

then for X_2

$$\begin{aligned} f(\mathbf{X}) &= X_2 \wedge f_1^{X_2}(\mathbf{X}) \vee \neg X_2 \wedge f_1^{\neg X_2}(\mathbf{X}) \\ &= X_1 \wedge X_2 \wedge X_3 \vee X_1 \wedge \neg X_2 \wedge X_3 \vee \neg X_1 \wedge X_2 \wedge X_3 \end{aligned}$$

Shannon expansion for X_3 *does not change the DNF formula.*

We can notice that the BDD is not reduced, indeed the no-redundancy condition is violated in n_2'*. Removing the redundant node generates the following ROBDD*



10.4 Conclusions

In this chapter we illustrated the Decision Diagrams, which are directed acyclic graphs. In particular the Binary Decision Diagrams are of particular importance because they make mutually exclusive the disjuncts of a DNF propositional formula. For this reason they are used to perform probabilistic (exact) inference by systems such as ProbLog [30] and PITA [121, 122].

The next chapter presents the fundamentals ideas for probabilistic (exact) logical inference.

Chapter 11

Fundamentals of Exact Probabilistic Logical Inference

In this chapter we discuss methods for probabilistic logical inference and in particular approaches for exact inference. A brief overview of existing approaches for probabilistic logical inference is given in Section 11.1. Section 11.2 explains, in general terms, how to perform exact probabilistic logical inference and the theory behind it. In order to compute the probability of a query, it is necessary to make the explanations pairwise incompatible. Here we discuss two approaches to do that: the splitting algorithm, illustrated in Section 11.3 and approaches based on Binary Decision Diagrams (BDDs), discussed in Section 11.5. Section 11.6 draws conclusions.

11.1 Inference Approaches

The probabilistic logical inference problem can be divided into two categories: *exact inference* and *approximate inference*.

Exact Inference
The aim is to compute the exact probabilistic value of a query. There are several approach, this chapter illustrates some of them.

Lifted inference is an important subcategory of exact inference and a research field of growing interest. Lifted approaches perform exact probabilistic logical inference at the lifted, i.e. non-ground, level, this means that we treat individual as a whole and the inference is realized without grounding the model. Example 11.1.1 helps to clarify the usefulness of lifted inference.

Example 11.1.1
Consider the following ProbLog program

```
p :: drinks(X).
alcoholic(X) :- friends(X,Y),drinks(Y).
```

The probability of `barney` *being an alcoholic is* $P($`alcoholic(barney)`$) = 1-(1-p)^m$*, where* m *is number of friends of* `barney`*, the more friends* `barney` *has the higher the*

probability. This means it suffices to know how many friends `barney` *has to compute the probability that* `barney` *is an alcoholic. It is not necessary to know the identities of these friends, and thus there is no need to ground the clauses.*

During the last decade, various approaches for PLP lifted inference have been proposed. For instance Poole in [133] presented a lifted version of variable elimination, which is a standard method for graphical models to compute probabilistic inference. In [134] the authors modified the Prolog Factor Language [41] by adding two new operators and applied Lifted Variable Elimination [133]. Van den Broeck et al. in [135, 136, 137] proposed a different approach in which a program is transformed in a generalization of a d-DNNF and then weighted model counting is performed. Recent surveys on lifted inference can be found in [138] and [139].

Approximate Inference
This type of inference is used when one wants to reduce the cost of the inference process by computing an approximation of the value of probability. One approach is to take a sample of normal programs from the probabilistic program[1] and then count the normal programs where the query is valid. The probability is the ratio of the programs where the query succeeds to the size of the sample [49]. Another approach is to compute the lower and upper bounds of the probability [140, 141].

11.2 Exact Probabilistic Logical Inference

In this section we discuss some techniques for exact inference. In Chapters 6 and 9 we showed the equations for computing the exact probability of a query, for PLP and PDLs that follow DISPONTE respectively. We said that it can be done by summing the probabilities of the worlds where the query succeeds. However, we also mentioned that calculating the probability of a query by generating all possible worlds is infeasible, for this reason explanations are used (see Definition 6.5 for PLP and Definition 9.4 for PDLs that follow DISPONTE). We also said that, given a query Q, if a covering set of composite choices K is pairwise incompatible, then the probability of Q is equal to the sum of the probabilities of the composite choices (see Equation (6.7)). Now, *the set of all the explanations of Q is a covering set*. Unfortunately, in general, explanations are not mutually exclusive, i.e. pairwise incompatible. However, the following results obtained by Poole in [142] could help us.

Definition 11.1 Split of a set of composite choices [142]
Let $\alpha_\theta = \{(C, \theta, 1), \ldots, (C, \theta, n)\}$ the set of the possible atomic choices for the clause C given the grounding substitution θ, where n is the number of heads (including, possibly, the *null* head); and let κ be a composite choice and C be a clause such that $\kappa \cap \alpha_\theta = \emptyset$, the *split* of κ on $C\theta$ is the set of composite choices $S_{\kappa,C\theta} = \{\kappa \cup \{(C, \theta, 0)\}, \kappa \cup \{(C, \theta, 1)\}, \ldots, \kappa \cup \{(C, \theta, n)\}\}$. It is important to notice that κ and $S_{\kappa,C\theta}$ identify the same set of possible worlds, i.e., that $\omega_\kappa = \omega_{S_{\kappa,C\theta}}$. □

[1] If you remember, the *distribution semantics* defines a distribution over normal logic programs, called worlds (Chapter 5).

If we are working with PDLs under DISPONTE an atomic choice is a couple of the form (C_i, k), where C_i is an axiom and $k \in \{0, 1\}$. Atomic choices for PDLs do not have a substitution, as we include or exclude axioms as a whole instead of their instantiations. To perform the split operation in PDLs, we just have to ignore substitutions.

The following theorem is extremely important in order to compute the probability of a query.

Theorem 11.1 Splitting [142]
Given a finite set K of finite composite choices, there exists a finite set K' of pairwise incompatible finite composite choices such that K and K' are equivalent.

Proof. Given a finite set of finite composite choices K, in order to form a new set K' of composite choices equivalent to K, two possibilities are given:

1. **removing dominated elements**: if $\kappa_1, \kappa_2 \in K$ and $\kappa_1 \subset \kappa_2$, let $K' = K \setminus \{\kappa_2\}$.

2. **splitting elements**: if $\kappa_1, \kappa_2 \in K$ are compatible and neither is a superset of the other, there is a $(C, \theta, k) \in \kappa_1 \setminus \kappa_2$. We replace κ_2 by the split of κ_2 on $c\theta$. Let $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2, C\theta}$.

In both cases $\omega_K = \omega_{K'}$. Since K is a finite set of finite composite choices, these two operations can be repeatedly executed until no one can be applied. The resulting set K' is pairwise incompatible and is equivalent to the original set. □

Thanks to Theorem 11.1, if we are able to obtain an equivalent set of pairwise incompatible explanations K' from the set of covering explanations K for the query Q, then we can use Equation 6.7 to calculate the probability of the query.

The proof of Theorem 11.1, is the basis of an algorithm, called **splitting algorithm**, presented in Section 11.3, which it is known to terminate.

Theorem 11.2 Probability of two equivalent pairwise incompatible sets of composite choices [143, 142]
If K_1 and K_2 are both pairwise incompatible finite sets of finite composite choices such that they are equivalent then $P(K_1) = P(K_2)$.

Another important result is given by the following theorem

Theorem 11.3 Probability of two pairwise incompatible sets of composite choices [3]
Given two finite sets of finite composite choices K_1 and K_2, if $K_1 \subseteq K_2$, then $P(K_1) \leq P(K_2)$.

Theorem 11.3 is really interesting because it tells us that, even if we find a set of explanations K that does not cover a query Q, $P(K)$ represents a lower bound of the exact probability of the query $P(Q)$.

The problem of calculating the probability of a query is therefore reduced to that of obtaining a covering set of explanations and then making it pairwise incompatible.

We can use different techniques to obtain the explanations for queries. One method is by means of (a possibly modified version of) SLDNF-resolution, This technique is used for ICL and LPADs by the PITA and `cplint` systems [27, 144] and by ProbLog1 [45, 140]. Another one is to use the HST algorithm with the tableau algorithm. Once a covering set K of explanations for a query Q have been obtained, to make them mutually incompatible, two approaches are usually exploited:

1. Splitting algorithm.
2. We can associate Boolean variables with axioms and define a DNF Boolean formula $f_K(\mathbf{X})$, in which every disjunct represents an explanation. Then translate the formula to a target language that makes the disjuncts mutually exclusive.

The splitting algorithm is illustrated in detail in Section 11.3. For the latter approach, do we know a language that makes the disjuncts of a DNF formula mutually exclusive? Yes, Decision Diagrams! This language was found to give good performances and it is discussed in detail in Section 11.5.

Alternatively to explanation finding approaches, we can use algorithms that compute the *pinpointing formula*.

11.3 Splitting Algorithm

From the proof of Theorem 11.1, the **splitting algorithm**, shown in Algorithm 11.1 can be built by looping the two operations.

Algorithm 11.1 Splitting Algorithm.

```
 1: procedure SPLIT(K)
 2:     Input: set of composite choices K
 3:     Output: pairwise incompatible set of composite choices equivalent to K
 4:     loop
 5:         if ∃κ1, κ2 ∈ K and κ1 ⊂ κ2 then
 6:             K ← K \ {κ2}
 7:         else
 8:             if ∃κ1, κ2 ∈ K compatible then
 9:                 choose (C, θ, k) ∈ κ1 \ κ2
10:                 Let S_{κ2,Cθ} be the split of κ2 on Cθ
11:                 K ← K \ {κ2} ∪ S_{κ2,Cθ}
12:             else
13:                 exit and return K
14:             end if
15:         end if
16:     end loop
17: end procedure
```

The lines 5-6 correspond to the first operation of the proof, whereas the second operation correspond to the pseudo-code in lines 8-11. If no operation can be applied the algorithm exits (line 13). AILog2 [145] is a system based on the splitting algorithm which can perform probabilistic logic reasoning on Independent Choice Logic (ICL).

The following example shows a case where it is possible to use the splitting algorithm for an LPAD.

Example 11.3.1 Splitting algorithm for LPADs
Consider the following LPAD "Crime and Punishment"

$$
\begin{aligned}
&\texttt{nihilist(X) :- killed(X,Y).}\\
C_1 = &\texttt{nihilist(X) : 0.3 :- student(X).}\\
C_2 = &\texttt{great\_man(X) : 0.2 :- nihilist(X).}\\
C_3 = &\texttt{killed(ras,aly) : 0.6.}\\
C_4 = &\texttt{killed(ras,liz) : 0.5.}\\
&\texttt{student(ras).}
\end{aligned}
$$

*This program states that if you killed someone then you are a nihilist, the students are nihilist with probability 0.3, whoever is a nihilist is a "great man" with probability 0.2 and Raskolnikov (*ras*) killed Alyona (*aly*) and Lizaveta (*liz*) with probability 0.6 and 0.5 respectively.*

A covering set of explanations for the query $Q =$ great_man(ras) *is*

$$
\begin{aligned}
K &= \{\kappa_1, \kappa_2, \kappa_3\}\\
\kappa_1 &= \{(C_1, \{X/\texttt{ras}\}, 1), (C_2, \{X/\texttt{ras}\}, 1)\}\\
\kappa_2 &= \{(C_3, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1)\}\\
\kappa_3 &= \{(C_4, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1)\}
\end{aligned}
$$

The explanations κ_1 and κ_2 are compatible so we can apply the second operation of splitting algorithm (Algorithm 11.1) (lines 8-11), obtaining

$$
\begin{aligned}
K_1 &= \{\kappa_1, \kappa_2', \kappa_2'', \kappa_3\}\\
\kappa_1 &= \{(C_1, \{X/\texttt{ras}\}, 1), (C_2, \{X/\texttt{ras}\}, 1)\}\\
\kappa_2' &= \{(C_3, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1), (C_1, \{X/\texttt{ras}\}, 0)\}\\
\kappa_2'' &= \{(C_3, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1), (C_1, \{X/\texttt{ras}\}, 1)\}\\
\kappa_3 &= \{(C_4, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1)\}
\end{aligned}
$$

Now $\kappa_1 \subset \kappa_2''$, i.e. κ_1 dominates κ_2'', therefore we can apply the first operation of the splitting algorithm (lines 5-6) and remove κ_2'', obtaining

$$K_2 = \{\kappa_1, \kappa_2', \kappa_3\}$$

The explanations κ_1 and κ_3 are compatible so we apply again the second operation of splitting algorithm, obtaining

$$
\begin{aligned}
K_3 &= \{\kappa_1, \kappa_2', \kappa_3', \kappa_3''\}\\
\kappa_1 &= \{(C_1, \{X/\texttt{ras}\}, 1), (C_2, \{X/\texttt{ras}\}, 1)\}\\
\kappa_2' &= \{(C_3, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1), (C_1, \{X/\texttt{ras}\}, 0)\}\\
\kappa_3' &= \{(C_4, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1), (C_1, \{X/\texttt{ras}\}, 0)\}\\
\kappa_3'' &= \{(C_4, \emptyset, 1), (C_2, \{X/\texttt{ras}\}, 1), (C_1, \{X/\texttt{ras}\}, 1)\}
\end{aligned}
$$

We can remove κ_3'' because is dominated by κ_1

$$K_4 = \{\kappa_1, \kappa_2', \kappa_3'\}$$

Now we have that the explanations κ_2' and κ_3' are compatible so we apply again the second operation, obtaining

$$\begin{aligned}
K_5 &= \{\kappa_1, \kappa_2', \kappa_3''', \kappa_3''''\} \\
\kappa_1 &= \{(C_1, \{\mathsf{X}/\mathtt{ras}\}, 1), (C_2, \{X/\mathtt{ras}\}, 1)\} \\
\kappa_2' &= \{(C_3, \emptyset, 1), (C_2, \{X/\mathtt{ras}\}, 1), (C_1, \{\mathsf{X}/\mathtt{ras}\}, 0)\} \\
\kappa_3''' &= \{(C_4, \emptyset, 1), (C_2, \{X/\mathtt{ras}\}, 1), (C_1, \{\mathsf{X}/\mathtt{ras}\}, 0), (C_3, \emptyset, 0)\} \\
\kappa_3'''' &= \{(C_4, \emptyset, 1), (C_2, \{X/\mathtt{ras}\}, 1), (C_1, \{\mathsf{X}/\mathtt{ras}\}, 0), (C_3, \emptyset, 1)\}
\end{aligned}$$

We can remove κ_3'''' because is dominated by κ_2'

$$K_6 = \{\kappa_1, \kappa_2', \kappa_3'''\}$$

Our covering set of explanations is now pairwise incompatible, we can finally compute the probability of the query

$$P(Q) = P(K_6) = P(\kappa_1) + P(\kappa_2') + P(\kappa_3''') = 0.3 \cdot 0.2 + 0.6 \cdot 0.2 \cdot 0.7 + 0.5 \cdot 0.2 \cdot 0.7 \cdot 0.4 = 0.172$$

In the next example we have a knowledge base under DISPONTE and we show another example of application of the splitting algorithm.

Example 11.3.2 Splitting algorithm for Probabilistic Description Logics *Consider another example inspired by the* `people+pets` *ontology proposed in [102]*

$$\begin{aligned}
&\mathsf{DogOwner} \sqsubseteq \mathsf{PetOwner} \\
&\mathsf{CatOwner} \sqsubseteq \mathsf{PetOwner} \\
E_1 &= 0.6 :: \mathsf{kevin} : \mathsf{DogOwner} \\
E_2 &= 0.6 :: \mathsf{kevin} : \mathsf{CatOwner} \\
E_3 &= 0.7 :: \mathsf{PetOwner} \sqsubseteq \mathsf{Ecologist}
\end{aligned}$$

The KB indicates that the individual kevin *owns a dog with probability 0.6, the same individual owns a cat with probability 0.6 and pet owners are ecologists with probability 0.7.*

A covering set of explanations for the query $Q =$ kevin : Ecologist *is*

$$\begin{aligned}
K &= \{\kappa_1, \kappa_2\} \\
\kappa_1 &= \{(E_2, 1), (E_3, 1)\} \\
\kappa_2 &= \{(E_1, 1), (E_3, 1)\}
\end{aligned}$$

The explanations are compatible so we can apply the second operation of splitting algorithm (Algorithm 11.1) (lines 8-11), obtaining

$$\begin{aligned}
K_1 &= \{\kappa_1, \kappa_2', \kappa_2''\} \\
\kappa_1 &= \{(E_2, 1), (E_3, 1)\} \\
\kappa_2' &= \{(E_1, 1), (E_3, 1), (E_2, 0)\} \\
\kappa_2'' &= \{(E_1, 1), (E_3, 1), (E_2, 1)\}
\end{aligned}$$

Now $\kappa_1 \subset \kappa_2''$, i.e. κ_1 dominates κ_2'', therefore we can apply the first operation of the splitting algorithm (lines 5-6). The resulting pairwise incompatible set of composite choices that covers Q are

$$K_2 = \{\kappa_1, \kappa_2'\}$$

Therefore the probability of Q is

$$P(Q) = P(K_2) = P(\kappa_1) + P(\kappa_2') = 0.6 \cdot 0.7 \cdot 0.4 + 0.6 \cdot 0.7 = 0.588$$

11.4 Inference with Multi-valued Decision Diagrams

Let C_i be the clauses/axioms of a knowledge base, we can define the following associations:

$$C_i\theta_j \leftrightarrow \text{multivalued random variable } X_{ij}$$
$$(C_i, \theta_j, k) \leftrightarrow \text{assignment } X_{ij} = k, k \in \{1, \ldots, n_i\}$$

In this way, given a set of covering explanations K, we obtain the following DNF formula f_K

$$f_K(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(C_i,\theta_j,k) \in \kappa} X_{ij} = k \tag{11.1}$$

The disjuncts in the formula are not necessarily mutually exclusive, the probability of the query can not be computed by a summation as in Equation (6.7). The problem of computing the probability of a Boolean formula in DNF, known as *disjoint sum*, is #P-complete [146].

We can apply *knowledge compilation* [147] to the Boolean formula $f_K(\mathbf{X})$ in order to translate it into a "target language" that allows the computation of its probability in polynomial time. We can use decision diagrams as a target language. Since the random variables appearing in the Boolean formula that are associated with atomic choices can take on multiple values, we need to use Multivalued Decision Diagrams (MDDs) [123].

As mentioned in Chapter 10 an MDD splits its paths on the basis of the values of a variable, the branches are mutually exclusive and a dynamic programming algorithm can be applied for computing the probability [45].

Example 11.4.1 MDD of a query
Consider the same LPAD of Example 6.2.1

C_1 = `epidemic : 0.6; pandemic : 0.3 :- flu(X),cold.`
C_2 = `cold : 0.7.`
`flu(david).`
`flu(robert).`

Clause C_1 has two groundings: $C_1\theta_1$ with $\theta_1 = \{$`X/david`$\}$ and $C_1\theta_2$ with $\theta_2 = \{$`X/robert`$\}$. Clause C_2, instead, has only one grounding $C_2\emptyset$.

We can make the following associations:

$$C_1\theta_1 \leftrightarrow X_{11}$$
$$C_1\theta_2 \leftrightarrow X_{12}$$
$$C_2\emptyset \leftrightarrow X_{21}$$

X_{11} and X_{12} can take three values since C_1 has three possible heads: `epidemic`, `pandemic` *and null, with indices 1, 2 and 3 respectively, whereas X_{21} can take only two values since C_2 has only two heads:* `cold`, *null, with indices 1 and 2 respectively. A possible set of covering explanations for the query $Q =$* `epidemic.` *is (from Example 6.2.3)*

$$K = \{\kappa_1, \kappa_2\}$$
$$\kappa_1 = \{(C_1, \{\texttt{X/david}\}, 1), (C_2, \emptyset, 1)\}$$
$$\kappa_2 = \{(C_1, \{\texttt{X/robert}\}, 1), (C_2, \emptyset, 1)\}$$

Each atomic choice can be associated with the propositional equation $X_{ij} = k$. Using Equation (11.1), the query is true if the following DNF formula is true:

$$f_K(\mathbf{X}) = (X_{21} = 1 \wedge X_{11} = 1) \vee (X_{21} = 1 \wedge X_{12} = 1) \tag{11.2}$$

Figure 11.1 shows the MDD corresponding to Equation (11.2).

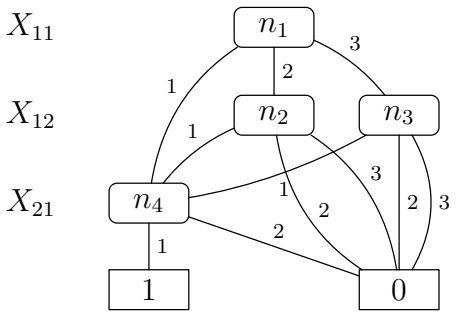


Figure 11.1: MDD corresponding to Equation (11.2).

Unfortunately, most packages for the manipulation of decision diagrams are restricted to work on BDDs. These packages offer Boolean operators among BDDs and apply simplification rules to the results of operations in order to reduce as much as possible the size of the binary decision diagram, obtaining a reduced BDD[2].

11.5 Inference with Binary Decision Diagrams

To work on MDDs with a BDD package we must represent multi-valued variables by means of binary variables. The following encoding, used in [148], gives good performance. For a multi-valued variable X_{ij}, corresponding to a ground clause $C_i\theta_j$, having

[2]In Chapter 10 we showed that the order of the variables affect the size of the BDD. Therefore, it is important to have BDD packages that have a good heuristics to order the variables.

n_i values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. Binary Decision Diagrams obtained in this way can be used as well for computing the probability of queries by associating a parameter π_{ik} with each Boolean variable X_{ijk}, representing $P(X_{ijk} = 1)$. The parameters are obtained from those of multi-valued variables in this way:

$$\begin{aligned} &\pi_{i1} = \Pi_{i1}, \\ &\ldots, \\ &\pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1 - \pi_{ij})} \\ &\ldots \end{aligned}$$

up to $k = n_i - 1$, where Π_{ik} is the probabilistic value assigned to the k-th head atom of the i-th clause.

Using the above transformation, we can now translate the DNF formula f_K of Equation (11.1) into a BDD. The problem of compiling a Boolean formula into the smallest BDD is NP-hard [149]. However, Riguzzi in [27] experimentally showed that inference approaches based on BDDs are faster than those based on the splitting algorithm[3].

As mentioned in Section 10.3 BDDs make the disjuncts, and hence the explanations, pairwise incompatible. Once we have obtained the BDD from the DNF Boolean formula f_K, to compute the probability of the query, we can use function PROB [3, 92, 91] shown in Algorithm 11.2. This algorithm traverses the diagram from the leaves to the root and computes the probability of a formula encoded as a BDD.

When a node is visited, its value is stored in a table so that, when the same node is visited again, its probability can be retrieved from the table. This optimization is necessary to reach linear cost in the number of nodes. Without it the cost of the function PROB would be proportional to 2^n where n is the number of Boolean variables.

If instead of solving the MIN-A-ENUM problem our reasoner computes the pinpointing formula, we can't use the splitting algorithm directly. In order to obtain a covering set of composite choices, we have to convert the pinpointing formula into a DNF formula and removing disjuncts implying other disjuncts. However, it is well-known that this can cause an exponential blowup. It makes more sense to directly use BDDs.

The following are some inference examples by means of BDDs.

Example 11.5.1 BDD of the query `epidemic`. (Example 11.4.1 cont.) *We consider the covering set of explanations obtained in Example 11.4.1 for the query $Q =$ `epidemic`. We convert each of the 3-valued variables X_{11} and X_{12} into two Boolean variables, X_{111} and X_{112} for X_{11}, and X_{121} and X_{122} for X_{12}. X_{21} is a 2-valued variable and is converted into the Boolean variable X_{211}. Equation 11.2 can be converted into the equivalent function*

$$f'_K(\mathbf{X}) = (X_{111} \wedge X_{211}) \vee (X_{121} \wedge X_{211}) \tag{11.3}$$

The equivalent BDD with order $X_{111} \prec X_{121} \prec X_{211}$ is shown in Figure 11.2.

[3] In [27], the author compared a system called PICL, based on BDDs, with AILog2 which uses the splitting algorithm.

Algorithm 11.2 Function PROB

```
1: function PROB(node, nodesTab)
2:     Input: a BDD node node
3:     Input: a table containing the probability of already visited nodes nodesTab
4:     Output: the probability of the Boolean function associated with the node
5:     if node is a terminal then
6:         return value(node)                          ▷ value(node) is 0 or 1
7:     else
8:         scan nodesTab looking for node
9:         if found then
10:            let P(node) be the probability of node in nodesTab
11:            return P(node)
12:        else
13:            let X be v(node)                ▷ v(node) is the variable associated with node
14:            P_1 ←PROB(child_1(node))
15:            P_0 ←PROB(child_0(node))
16:            P(node) ← P(X) · P_1 + (1 − P(X)) · P_0
17:            add the pair (node,P(node)) to nodesTab
18:            return P(node)
19:        end if
20:    end if
21: end function
```

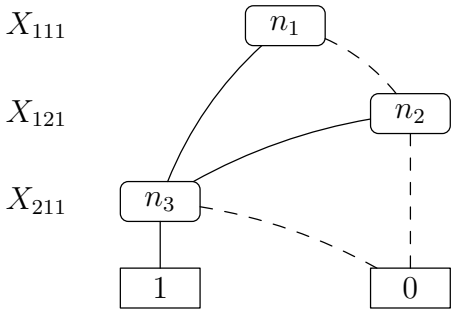


Figure 11.2: BDD for Example 11.5.1 equivalent to the MDD in Figure 11.1.

If we apply function PROB *in Algorithm 11.2*

$$\begin{aligned} \text{PROB}(n_3) &= 0.7 \cdot 1 + 0.3 \cdot 0 = 0.7 \\ \text{PROB}(n_2) &= 0.6 \cdot 0.7 + 0.4 \cdot 0 = 0.42 \\ P(Q) = \text{PROB}(n_1) &= 0.6 \cdot 0.7 + 0.4 \cdot 0.42 = 0.588 \end{aligned}$$

Example 11.5.2
Consider the LPAD in Example 11.3.1 and the covering set of explanations K for the query $Q =$ `great_man(ras)` *obtained in the same example.*

All the probabilistic clauses have only one grounding: $C_1\theta_1$, $C_2\theta_2$, $C_3\emptyset$ and $C_4\emptyset$, with $\theta_1 = \theta_2 = \{$`X/ras`*$\}$.*

We can make the following associations:

$$C_1\theta_1 \leftrightarrow X_{11}$$
$$C_2\theta_2 \leftrightarrow X_{21}$$
$$C_3\emptyset \leftrightarrow X_{31}$$
$$C_4\emptyset \leftrightarrow X_{41}$$

If we set that the null head has index $k = 2$ in all the triples (C_i, θ_j, k). Using Equation (11.1), the query Q is true if the following DNF formula is true:

$$f_K(\mathbf{X}) = (X_{11} = 1 \wedge X_{21} = 1) \vee (X_{31} = 1 \wedge X_{21} = 1) \vee (X_{41} = 1 \wedge X_{21} = 1)$$

We convert each of the 2-valued variables X_{11}, X_{21}, X_{31} and X_{41} into the Boolean variables, X_{111}, X_{211}, X_{311} and X_{411}. f_K can be converted into the equivalent following formula with Boolean variables

$$f'_K(\mathbf{X}) = (X_{111} \wedge X_{211}) \vee (X_{311} \wedge X_{211}) \vee (X_{411} \wedge X_{211}) \tag{11.4}$$

Figure 11.3 shows the BDD corresponding to Equation (11.4) with variable order $X_{311} \prec X_{211} \prec X_{411} \prec X_{111}$.

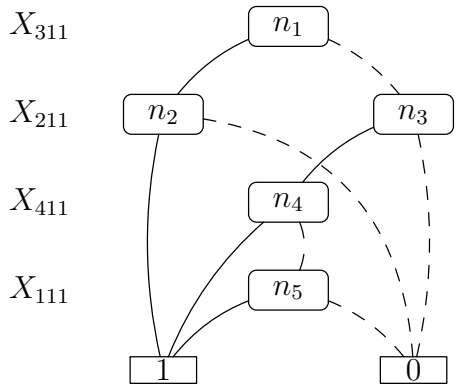


Figure 11.3: BDD for Example 11.5.2.

By applying function Prob *in Algorithm 11.2, we obtain*

$$\begin{aligned}
\textsc{Prob}(n_5) &= 0.3 \cdot 1 + 0.7 \cdot 0 = 0.3 \\
\textsc{Prob}(n_4) &= 0.5 \cdot 1 + 0.5 \cdot 0.3 = 0.65 \\
\textsc{Prob}(n_3) &= 0.2 \cdot 0.65 + 0.8 \cdot 0 = 0.13 \\
\textsc{Prob}(n_2) &= 0.2 \cdot 1 + 0.8 \cdot 0 = 0.2 \\
P(Q) = \textsc{Prob}(n_1) &= 0.6 \cdot 0.2 + 0.4 \cdot 0.13 = 0.172
\end{aligned}$$

This result is equivalent to the one obtained in Example 11.3.1.

If we are using PDLs under DISPONTE we can use BDDs directly.

Example 11.5.3
Let us consider the same KB in Example 11.3.2, instead of using the splitting algorithm, we can use a BDD to compute the probability of the query $Q = \mathsf{kevin} : \mathsf{Ecologist}$.

Given the following covering set of explanations

$$K = \{\kappa_1, \kappa_2\}$$
$$\kappa_1 = \{(E_2, 1), (E_3, 1)\}$$
$$\kappa_2 = \{(E_1, 1), (E_3, 1)\}$$

If we associate the Boolean random variables X_1 *to* E_1, X_2 *to* E_2 *and* X_3 *to* E_3, *we obtain the following DNF Boolean formula*

$$f_K(\mathbf{X}) = (X_1 \wedge X_3) \vee (X_2 \wedge X_3)$$

The equivalent BDD with order $X_1 \prec X_2 \prec X_3$ *is shown in Figure 11.4*

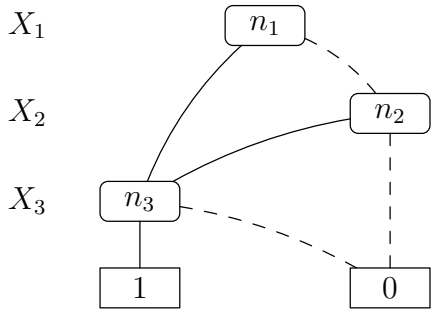


Figure 11.4: BDD for Example 11.5.3.

By applying function PROB *in Algorithm 11.2, we obtain*

$$\begin{aligned} \text{PROB}(n_3) &= 0.7 \cdot 1 + 0.3 \cdot 0 = 0.7 \\ \text{PROB}(n_2) &= 0.6 \cdot 0.7 + 0.4 \cdot 0 = 0.42 \\ P(Q) = \text{PROB}(n_1) &= 0.6 \cdot 0.7 + 0.4 \cdot 0.42 = 0.588 \end{aligned}$$

The result is equal to the one obtained in Example 11.3.2

Example 11.5.4
Consider a slightly different knowledge base [3, 92]:

$$C_1 = \exists \mathsf{hasAnimal.Pet} \sqsubseteq \mathsf{NatureLover}$$
$$C_2 = (\mathsf{kevin,fluffy}) : \mathsf{hasAnimal}$$
$$C_3 = (\mathsf{kevin,tom}) : \mathsf{hasAnimal}$$
$$E_1 = 0.4 :: \mathsf{fluffy} : \mathsf{Dog}$$
$$E_2 = 0.3 :: \mathsf{tom} : \mathsf{Cat}$$
$$E_3 = 0.6 :: \mathsf{Cat} \sqsubseteq \mathsf{Pet}$$
$$E_4 = 0.5 :: \mathsf{Dog} \sqsubseteq \mathsf{Pet}$$

A covering set of explanations for the query axiom $Q = kevin : NatureLover$ is

$$K = \{\kappa_1, \kappa_2\}$$
$$\kappa_1 = \{(E_1, 1), (E_4, 1)\}$$
$$\kappa_2 = \{(E_2, 1), (E_3, 1)\}$$

If we associate the random variables X_1 to E_1, X_2 to E_2, and so on, we obtain the following DNF Boolean formula

$$f_K(\mathbf{X}) = (X_1 \wedge X_4) \vee (X_2 \wedge X_3)$$

If we chose the order $X_1 \prec X_2 \prec X_3 \prec X_4$ the BDD associated with the set K of explanations is shown in Figure 11.5.

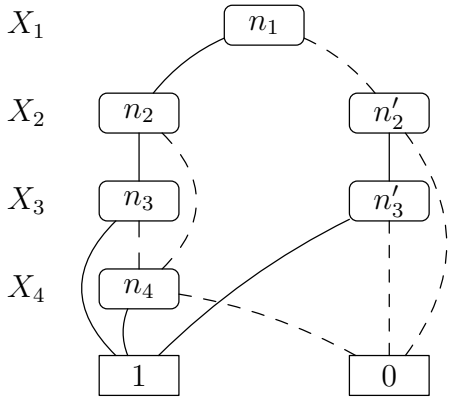


Figure 11.5: BDD for Example 11.5.4 with order $X_1 \prec X_2 \prec X_3 \prec X_4$.

By applying the function Prob *in Algorithm 11.2 we get*

$$\begin{aligned}
\textsc{Prob}(n_4) &= 0.5 \cdot 1 + 0.5 \cdot 0 = 0.5 \\
\textsc{Prob}(n_3) &= 0.6 \cdot 1 + 0.4 \cdot 0.5 = 0.8 \\
\textsc{Prob}(n_2) &= 0.3 \cdot 0.8 + 0.7 \cdot 0.5 = 0.59 \\
\textsc{Prob}(n_3') &= 0.6 \cdot 1 + 0.4 \cdot 0 = 0.6 \\
\textsc{Prob}(n_2') &= 0.3 \cdot 0.6 + 0.7 \cdot 0 = 0.18 \\
\textsc{Prob}(n_1) &= 0.4 \cdot 0.59 + 0.6 \cdot 0.18 = 0.344
\end{aligned}$$

so $P(Q) = \textsc{Prob}(n_1) = 0.344$.

The order of the variables for the BDD is not optimal, if we had chosen the order $X_1 \prec X_4 \prec X_2 \prec X_3$ we would have had the BDD in Figure 11.6 which is more compact than the BDD in Figure 11.5.

The size of the BDD affects the number of operations of the function Prob *in Algorithm 11.2, indeed if we apply* Prob *to the BDD in Figure 11.6, we obtain*

$$\begin{aligned}
\textsc{Prob}(n_4) &= 0.6 \cdot 1 + 0.4 \cdot 0 = 0.6 \\
\textsc{Prob}(n_3) &= 0.3 \cdot 0.6 + 0.7 \cdot 0 = 0.18 \\
\textsc{Prob}(n_2) &= 0.5 \cdot 1 + 0.5 \cdot 0.18 = 0.59 \\
\textsc{Prob}(n_1) &= 0.4 \cdot 0.59 + 0.6 \cdot 0.18 = 0.344
\end{aligned}$$

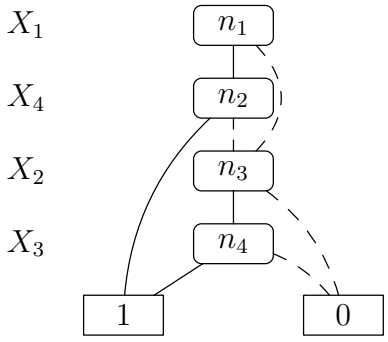


Figure 11.6: BDD for Example 11.5.4 with order $X_1 \prec X_4 \prec X_2 \prec X_3$.

so $P(Q) = \text{PROB}(n_1) = 0.344$.

11.6 Conclusions

In this chapter we discussed how to perform exact probabilistic logical inference. The problem of calculating the probability of a query consists of obtaining a covering set of explanations and then make them pairwise incompatible. We examined two approaches to make the explanations pairwise incompatible: the splitting algorithm and BDDs. Approaches based on BDDs are usually faster than approaches based on the splitting algorithm [27]. For this reason the systems for exact probabilistic inference presented in the next chapters are based on BDDs.

The next two chapters present several inference systems for PLP (Chapter 12) and for PDLs that follow DISPONTE (Chapter 13).

Chapter 12

Inference in Probabilistic Logic Programming

In this chapter we present the latest advances of the **cplint** system and its web interface **cplint on SWISH**. The chapter is organized as follows. After a brief introduction in Section 12.1, the `cplint` system and its main modules for (conditional) inference are presented in Section 12.2. Section 12.3 and Section 12.4 discuss causal reasoning and how to perform inference on hybrid probabilistic logic programs, i.e. probabilistic logic programs where some of the random variables are continuous, with `cplint`. `cplint`'s web interface, named `cplint` on SWISH, is described in Subsection 12.5.2. Section 12.6 illustrates related work. Finally Section 12.7 concludes the chapter.

12.1 Introduction

In Chapter 6 we introduced Probabilistic Logic Programming (PLP), this field of research aims to combine logic with probability theory, moreover we provided the theoretical foundations for the calculation of the probability of a query. In Chapter 11, instead, we presented two techniques for the calculation of the exact probability of a query. One based on splitting algorithm and one based on the use of BDDs. All the approaches seen so far are only theoretical, no real system that implements these approaches has been presented.

In [144] Riguzzi and Swift proposed a system called PITA that allows to perform exact inference. Later Riguzzi in [49] developed MCINTYRE a system for executing approximate inference by sampling. These two systems were gathered together into `cplint`.

The first part of this thesis tackled the problem of representing uncertain information by combining logic with probability. This part, instead, concerns reasoning over uncertain data. In this chapter we deal with reasoning in PLP. In particular, we present the latest features that we have developed for the `cplint` system and its web interface called `cplint` of SWISH.

12.2 cplint

cplint is a suite of programs for reasoning with LPADs. cplint contains modules for both inference and learning[1]. For inference we have two modules:

- the **PITA module** for exact inference [144], and
- the **MCINTYRE module** for approximate inference by sampling [49].

Below we illustrate these modules and how to use them.

12.2.1 Exact Inference: the PITA module

PITA [144] computes the probability of a query from a probabilistic program in the form of an LPAD by knowledge compilation [147]. PITA computes explanations for the query and encodes them using Binary Decision Diagrams (BDDs). Each explanation is a conjunction of equations of the form $Var = value$, where Var is a random variable associated with a ground clause and *value* is a possible value (the index of one of the atoms in the head). We can see that Var is multivalued, we could have used MDDs to make the explanations pairwise incompatible, but, as said before, many decision diagram packages only support BDDs.

PITA computes BDDs for explanations by transforming an LPAD into a normal program containing calls for manipulating BDDs. The idea is to add an extra argument to each subgoal to store a BDD encoding the explanations for the answers of the subgoal. The values of the extra argument of the subgoals are combined using a set of library functions:

- `init, end`: initialize and terminate the data structures for manipulating BDDs;
- `zero(-D), one(-D)`: return BDD `D` representing the Boolean constant 0 and 1;
- `and(+D1,+D2,-D0), or(+D1,+D2,-D0), not(+D1,-D0)`: Boolean operations between BDDs;
- `get_var_n(+R,+S,+Probs,-Var)`: returns the multi-valued random variable associated with rule `R` with grounding substitution `S` and list of probabilities `Probs`;
- `equality(+Var,+Value,-D)`: `D` is the BDD representing `Var=Value`, i.e. that the multivalued random variable *Var* is assigned *Value*;
- `ret_prob(+D,-P)`: returns the probability `P` of the BDD `D`.

In order to manage and manipulate BDDs we exploit the CUDD[2] (Colorado University Decision Diagram) library. The above functions are implemented in C as an interface to the CUDD library. A BDD is represented in Prolog as an integer that is a pointer in memory to the root node of the BDD.

The PITA transformation applies to atoms, literals, conjunctions of literals and clauses. The transformation for an atom `h` and a variable `D`, `PITA(h,D)`, is `h` with the

[1] For the learning modules see Chapter 15.

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

variable `D` added as the last argument. For the sake of simplicity, we consider here only positive literals, but the transformation can be applied also to negative literals (see [144]).

The transformation for a conjunction of literals $b_1,...,b_m$ is

$$\begin{aligned}&PITA(b_1,...,b_m,D)=one(DD_0),\\&\quad PITA(b_1,D_1),and(DD_0,D_1,DD_1),...,\\&\quad PITA(b_m,D_m),and(DD_{m-1},D_m,D).\end{aligned}$$

The disjunctive clause C_r = $h_1{:}\Pi_1;...;h_n{:}\Pi_n$:- $b_1,...,b_m$. where the parameters sum to 1, is transformed into the set of clauses `PITA(`C_r`)`:

$$\begin{aligned}&PITA(C_r,i) = PITA(h_i,D) :- PITA(b_1,...,b_m,DD_m),\\&\quad get\_var\_n(r,S,[\Pi_1,...,\Pi_n],Var),\\&\quad equality(Var,i,DD),and(DD_m,DD,D).\end{aligned}$$

for `i=1,...,n`, where `S` is a list containing all the variables appearing in C_r. If the parameters do not sum up to 1, the body is empty or the clause is non-disjunctive (a single head with probability 1), the transformation can be optimised.

We assume programs to be range restricted (see Definition 4.17). If the program is range restricted, when the goal `get_var_n(r,S,[`$\Pi_1,...,\Pi_n$`],Var)` is called, all the variables of the original clause, listed in `S`, are instantiated so `get_var_n/4` can associate a random variable with the instantiation of clause C_r.

The PITA transformation applied to clause C_1 of Example 6.2.1 yields

$$\begin{aligned}&PITA(C_1,1) = epidemic(D) :-\\&\quad one(DD_0),flu(X,D_1),and(DD_0,D_1,DD_1),\\&\quad cold(D_2),and(DD_1,D_2,DD_2),\\&\quad get\_var\_n(1,[X],[0.6,0.3,0.1],Var),\\&\quad equality(Var,1,DD),and(DD_2,DD,D).\\&PITA(C_1,2) = pandemic(D) :-\\&\quad one(DD_0),flu(X,D_1),and(DD_0,D_1,DD_1),\\&\quad cold(D_2),and(DD_1,D_2,DD_2),\\&\quad get\_var\_n(1,[X],[0.6,0.3,0.1],Var),\\&\quad equality(Var,2,DD),and(DD_2,DD,D).\end{aligned}$$

PITA is available for XSB Prolog [150], YAP Prolog [151] and SWI-Prolog [152].

The XSB version, the initial one, uses tabling, a logic programming technique that reduces computation time and ensures termination for a large class of programs [150]. The idea of tabling is simple: keep a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, the answers are retrieved from the store rather than recomputing them. Besides saving time, tabling ensures termination for programs without function symbols under the well-founded semantics [24].

PITA also uses a feature of XSB tabling called *answer subsumption* [150] that, when a new answer for a tabled subgoal is found, combines old answers with the new one according to a partial order or lattice. This feature is used to combine the BDDs that are built for different explanations of a goal, using `or/3` as the join operation of the lattice and `zero/1` as the predicate returning the bottom element of the lattice. For example, a unary predicate `p/1` must be declared as tabled by means of the declaration

`table p(_,or/3-zero/1).` If an answer `p(a,d1)` was found and a new answer `p(a,d2)` is derived, the answer `p(a,d1)` is replaced by `p(a,d3)`, where `d3` is obtained by calling `or(d1,d2,d3)`.

To compute the probability of a ground atom `A`, PITA uses predicate `prob/2` whose definition is

```
prob(A,Prob) :-
    add_bdd_arg(A,D,A1),
    call(A1),
    ret_prob(D,Prob).
```

where `add_bdd_arg(A,D,A1)` performs the PITA transformation `PITA(A,D)` for a literal `A` and a variable `D`, and `A1` contains the transformed literal. Since YAP and SWI-Prolog do not have answer subsumption in their tabling implementation, the collection of the various explanations for the goal is performed explicitly with this definition of `prob/2`:

```
prob(A,Prob) :-
    add_bdd_arg(A,D,A1),
    findall(D,A1,L),
    zero(Zero),
    foldl(or,L,Zero,DD),
    ret_prob(DD,Prob).
```

where `foldl/4` implements the higher order functional programming fold function and is available in the `apply` library of YAP and SWI-Prolog.

12.2.1.1 Conditional Exact Inference

To compute the probability of a conjunction of ground goals `G` given another conjunction of ground goals `E`, two clauses are added to the knowledge base:

$$\$goal(\texttt{D}) \texttt{ :- PITA(G,D).}$$

$$\$ev(\texttt{D}) \texttt{ :- PITA(E,D).}$$

and the queries $\$goal(\texttt{DG})$ and $\$ev(\texttt{DE})$ are asked. `DG` will contain the BDD representing the explanations for the goal and `DE` the BDD representing the explanations for the evidence. Then the conjunction of `DG` and `DE` is computed obtaining `DGE`. The probability to be returned is the fraction of the probability of `DGE` over the probability of `DE`, as shown in Algorithm 12.1.

12.2.2 Approximate Inference: the MCINTYRE module

MCINTYRE [49] performs approximate inference by sampling. It first transforms the program and then queries the transformed program. The disjunctive clause

$$\texttt{C}_i \texttt{ = h}_{i1}\texttt{:}\Pi_{i1}\texttt{;...;h}_{in}\texttt{:}\Pi_{in_i} \texttt{ :- b}_{i1}\texttt{,...,b}_{im_i}\texttt{.}$$

where the parameters sum to 1, is transformed into the set of clauses $\texttt{MC(C}_i\texttt{)}$:

Algorithm 12.1 Algorithm for computing the conditional probabilities.

1: **function** PROB(KB, G, E) ▷ Program KB, goal G, evidence E
2: Add $\$goal$(D) `:- PITA(G,D).` to KB
3: Add $\$ev$(D) `:- PITA(E,D).` to KB
4: Ask the queries $\$goal$(DG) and $\$ev$(DE)
5: DGE ← `bdd_and(DG,DE)`
6: PGE ← `ret_prob(DGE)`
7: PE ← `ret_prob(DE)`
8: **return** PGE/PE
9: **end function**

$$\begin{aligned}
&\texttt{MC(C}_i\texttt{,1) =} && \texttt{h}_{i1}\texttt{ :- b}_{i1}\texttt{,...,b}_{im_i}\texttt{,}\\
&&& \texttt{sample_head(ParList,i,S,NH),NH=1.}\\
&\texttt{...}\\
&\texttt{MC(C}_i\texttt{,}n_i\texttt{) =} && \texttt{h}_{in_i}\texttt{ :- b}_i\texttt{1,...,b}_{im_i}\texttt{,}\\
&&& \texttt{sample_head(ParList,i,S,NH),NH=}n_i\texttt{.}
\end{aligned}$$

where S is a list containing each variable appearing in $\texttt{C}_i$ and ParList is $[\Pi_{i1},\ldots,\Pi_{in_i}]$. If the parameters do not sum up to 1, the last clause (the one for *null*) is omitted. Basically, we create a clause for each head and we sample a head index at the end of the body with `sample_head/4`. If this index coincides with the head index, the derivation succeeds, otherwise it fails. Thus failure can occur either because one of the body literals fails or because the current clause is not part of the sample.

For example, clause C_1 of Example 6.2.1 becomes

```
MC(C1,1) = epidemic :- flu(X),cold,
           sample_head([0.6,0.3,0.1],1,[X],NH),NH=1.
MC(C1,2) = pandemic :- flu(X),cold,
           sample_head([0.6,0.3,0.1],1,[X],NH),NH=2.
```

The predicate `sample_head/4` samples an index from the head of a clause and uses the built-in Prolog predicates `recorded/3` and `recorda/3` for respectively retrieving or adding an entry to the internal database. Since `sample_head/4` is at the end of the body and since we assume the program to be range restricted, at that point all the variables of the clause have been grounded. If the rule instantiation had already been sampled, `sample_head/4` retrieves the head index with `recorded/3`, otherwise it samples a head index with `sample/2`:

```
sample_head(_ParList,R,VC,NH):-
    recorded(exp,(R,VC,N),_),!,
    NH=N.
sample_head(ParList,R,VC,N):-
    sample(ParList,NH),
    recorda(exp,(R,VC,NH),_),
    N=NH.

sample(ParList, HeadId) :-
```

```
        random(Prob),
        sample(ParList, 0, 0, Prob, HeadId).
    sample([HeadProb|Tail], Index, Prev, Prob, HeadId) :-
        Succ is Index + 1,
        Next is Prev + HeadProb,
        (Prob =< Next ->
            HeadId = Index
            ;
            sample(Tail, Succ, Next, Prob, HeadId)
        ).
```

Tabling can be effectively used to speed up the computation. Before executing a new goal the previous tables obtained by sampling and the previous samples must be removed. To sample a truth value for a ground atom `Goal` from the program we use the following predicate

```
sample(Goal) :-
    abolish_all_tables,
    erase_samples,
    call(Goal).
```

To compute the probability of a query, a number N of samples is taken and the probability is given by S/N where S is the number of times that `sample/1` succeeds.

12.2.2.1 Conditional Approximate Inference

Similarly to PITA, to compute the probability of a conjunction of ground goals `G` given another conjunction of ground goals `E`, two clauses are added to the knowledge base:

$$\$goal \texttt{ :- G.}$$

$$\$ev \texttt{ :- E.}$$

Conditional inference in MCINTYRE can be performed by means of rejection sampling or by Metropolis-Hastings Markov Chain Monte Carlo (MCMC) [7].

Rejection Sampling
In rejection sampling [153], you take a sample by first querying the evidence (with `sample(`$ev`)`) and, if the query is successful, query the goal in the same sample (with `sample(`$goal`)`), otherwise the sample is discarded. Rejection sampling is the easiest approach to realize, but it is also very slow.

To submit a conditional query using rejection sampling, you can use the predicate

```
mc_rejection_sample(:Query:atom,:Evidence:atom,+Samples:int,
   -Probability:float)
```

or

```
mc_rejection_sample(:Query:atom,:Evidence:atom,+Samples:int,
   -Successes:int,-Failures:int,-Probability:float).
```

Metropolis-Hastings MCMC
In Metropolis-Hastings MCMC [154], a Markov chain is built by taking an initial sample and by generating successor samples. A sample corresponds to a composite choice, which in turn corresponds to a set of worlds (see Chapter 6). The initial sample κ_0 is built with a meta-interpreter that randomly samples the choices so that the evidence is true. A successor sample κ is obtained by deleting a fixed number of sampled probabilistic choices, i.e. $\kappa_0 \supset \kappa$. Then the evidence is queried by taking a sample κ' starting with the undeleted choices with $\kappa \subset \kappa'$. If the query succeeds, the goal is queried by taking a sample κ'' with $\kappa' \subset \kappa''$, otherwise κ' is discarded. The sample is accepted with a probability of $\min\{1, \frac{|\kappa|}{|\kappa''|}\}$ where $|\kappa|$ is the number of choices (i.e. atomic choices) sampled in the previous sample and $|\kappa''|$ is the number of choices sampled in the current sample. Then the number of successes of the query is increased by 1 if the query succeeded in the last accepted sample. The final probability is given by the number of successes over the number of samples.

To perform a conditional query using Metropolis-Hastings MCMC, the available predicate is

```
mc_mh_sample(:Query:atom,:Evidence:atom,+Samples:int,+Lag:int,
 -Successes:int,-Failures:int,-Probability:float).
```

or

```
mc_mh_sample(:Query:atom,:Evidence:atom,+Samples:int,+Lag:int,
   -Successes:int,-Failures:int,-Probability:float).
```

12.3 Causal Inference with **cplint**

The study of causation was connected to graphical models by Pearl [155], even if diagrams were already used to represent causal models as early as the 1920's [156]. Graphical models are used to describe domains characterized by a set of random variables. Bayesian networks, in particular, are directed acyclic graphs where the variables are nodes and probabilistic dependences are represented as arcs: an arc from a node A to a node B means that A probabilistically influences B. An example of a Bayesian network is shown in Figure 12.1: it describes the domain of a medical study investigating the effects of a new drug on patients. The domain is described by three Boolean variables: Gender (F), Drug (C) and Recovery (E). Gender indicates the gender of the patient, Drug takes value 1 if the drug is administered to the patient under examination and value 0 if a placebo is administered, and Recovery whether the patient recovered from his illness. Gender influences Drug because the decision to administer or not the drug is taken on the basis of the sex of the patient. Gender and Drug influence Recovery because the outcome of the particular illness under examination depends on the sex of the patient and, hopefully, on the treatment.

Pearl [155] introduced *causal Bayesian networks*: these are Bayesian networks where an arc from a node A to a node B means that A directly causally influences B. Causal Bayesian networks can be used to perform causal reasoning, such as for example computing the effect of an action.

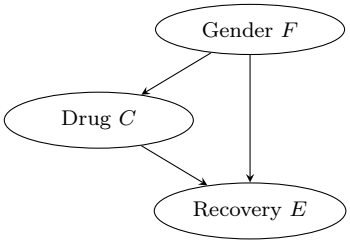


Figure 12.1: Bayesian network for a drug study domain.

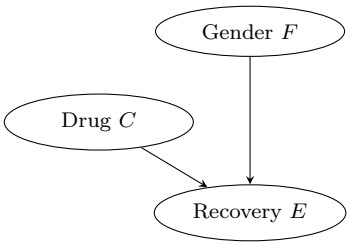


Figure 12.2: Mutilated version of the Bayesian network of Figure 12.1 for computing the effect of a drug.

An *action* or *intervention* in this context means setting a variable, say A, to a particular value, say a. The Bayesian network of Figure 12.1 is causal as we assume that the decision to administer or not the drug is taken on the basis of the sex of the patient. Moreover, the treatment and sex cause the patient to recover or not, as we assume that the illness depends on the gender.

In such a network one could for example ask what is the probability of recovery if we make the action of administering the drug, in other words what is the effect of the drug on recovery, the main aim of medical studies. This corresponds to computing the probability of $E = 1$ when setting C to 1. For answering such queries, Pearl shows that regular probabilistic reasoning cannot be used. So in this case computing $P(E = 1|C = 1)$ does not answer the question of what is the effect of the drug.

Pearl introduces a different calculus, called *do calculus*, to infer the effect of actions. In such a calculus, the action of setting a variable to a value is distinguished from the observation of that value for the variable. Actions appear inside a special *do* operator in the condition part of probabilistic queries. So to compute the effect of the drug on recovery, the query to answer is $P(E = 1|do(C = 1))$.

The *do* calculus reduces a query involving actions to a regular probabilistic query over a *mutilated* Bayesian network obtained by removing all incoming arcs from variables involved in actions. Then the query with actions as observations must be asked from the mutilated network. For example, to answer $P(E = 1|do(C = 1))$, the arc from Gender to Drug must be removed from the network of Figure 12.1 obtaining the network of Figure 12.2. Then the query $P(E = 1|C = 1)$ must be asked from the mutilated network. Note that there is no need to specify the conditional probability table (CPT) of the action variables (C in this case) in the mutilated network as the action variables are observed so the CPT does not influence the computation.

Equivalently, we can ask an unconditional query from the mutilated network where

the CPTs for the actions are set so that all the probability mass is assigned to the values set by the actions. For the example above, the conditional probability table of C would be given by $P(C = 1) = 1$ and $P(C = 0) = 0$ and the query would be $P(E = 1)$.

It is very important not to confound $P(E|do(C))$ with $P(E|C)$ because the results may be very different, as shown by the famous *Simpson's paradox*.

Example 12.3.1 Simpson's Paradox
From [155]:

> *Simpson's paradox* [...] *refers to the phenomenon whereby an event C increases the probability of E in a given population p and, at the same time, decreases the probability of E in every subpopulation of p. In other words, if F and $\neg F$ are two complementary properties describing two subpopulations, we might well encounter the inequalities*
>
> $$P(E|C) > P(E|\neg C)$$
>
> $$P(E|C, F) < P(E|\neg C, F)$$
> $$P(E|C, \neg F) < P(E|\neg C, \neg F)$$
>
> [...] *For example, if we associate C (connoting cause) with taking a certain drug, E (connoting effect) with recovery, and F with being a female, then* [...] *the drug seems to be harmful to both males and females yet beneficial to the population as a whole.*

Consider the situation exemplified by the following tables from [155]:

Combined	E	$\neg E$		RecoveryRate
Drug(C)	20	20	40	50%
Nodrug($\neg C$)	16	24	40	40%
	36	44	80	

Females	E	$\neg E$		RecoveryRate
Drug(C)	2	8	10	20%
Nodrug($\neg C$)	9	21	30	30%
	11	29	40	

Males	E	$\neg E$		RecoveryRate
Drug(C)	18	12	30	60%
Nodrug($\neg C$)	7	3	10	70%
	25	15	40	

As you can see taking the drug seems to be beneficial overall even if it is not for females and males.

The paradox derives because we must distinguish seeing from doing: we must distinguish observing *that the drug was administered from the* intervention *of administering*

the drug. The conditioning operator in probability calculus stands for "given that we see", whereas the do operator means "given that we do". So the do operator must be used to infer the effect of actions. If the model of the domain is the network from Figure 12.1, to compute $P(E = 1|do(C = 1))$ *and* $P(E = 1|do(C = 0))$ *we must compute* $P(E = 1|C = 1)$ *and* $P(E = 1|C = 0)$ *from the network of Figure 12.2 by using classical Bayesian inference. For these queries we get respectively 0.4 and 0.5, showing that the drug is not beneficial in the whole population exactly as it is not in the two subpopulations.*

Pearl's *do* calculus also deals with causal Bayesian networks where some of the variables are *unknown*, in the sense that we know that they exert an influence but they are not measurable so it is not possible to quantify this influence, i.e., we don't know how many they are and the CPTs where they are involved, we just know that some exist. When models contain such unknown variables, computing the effect of actions is not always possible, because we can't sum out the contribution of such variables since we don't know their number and CPTs. The *do* calculus provides rules for determining whether it is possible to compute the effect of an action even in the presence of unknown variables and to actually perform the computation. In `cplint` we consider only the *do* calculus for models with no unknown variables.

12.3.1 Causal Inference in Probabilistic Logic Programming

CP-logic [48] is a PLP language for causal reasoning whose semantics is based on probability trees that represent possible courses of events. The authors proved that their semantics is suitable for representing causation and the effects of causal laws. In particular, they highlighted that the inductive definitions of logic programming and the well-founded semantics of negation [24] produce models respecting most properties of causation, provided the program respects some weak constraints. The semantics of legal CP-logic programs coincides with that of LPADs, but there are LPADs that are not legal CP-theories, i.e., they cannot be assigned a causal semantics. However, these are corner cases in which the stratification level of a couple of atoms in a world is switched in a different world, so that it is not possible to establish a general stratification coherent with temporal precedence in all worlds.

The authors in [48] showed that the effect of actions in *do* calculus style can be computed from CP-theories when there are no unknown variables. In fact, clauses in CP-theories represent causal laws so in order to know the result of intervening on a single causal law, that law should be removed from the theory (and possibly replaced by a different law). For example, to compute the effects of an intervention that prevents a causal law C, that law must be removed from the theory. In case the intervention establishes a new causal law C', that law must be added to the theory. The modularity of CP-logic allows this.

Example 12.3.2
The computation of the effects of interventions is illustrated in [48] with a medical example:

A tumor in a patient's kidney might cause kidney failure, which might cause

the death of the patient; however, to make matters even worse, the tumor can also metastasize to the brain, which might also, independently, kill the patient. We can represent this as:

```
kidneyFailure : 0.1 :- kidneyTumor.
brainTumor : 0.1 :- kidneyTumor.
death : 0.5 :- brainTumor.
death : 0.9 :- kidneyFailure.
```

If we want to know what is the effect of putting the patient on a dialysis machine, which allows him to survive kidney failure, we can remove the last law and use the resulting theory for inference.

Starting from the results in [48] we modified the inference in `cplint` to allow the computation of the effect of actions of the form `do(A)` and `do(\+A)` where `A` is a ground literal. `do(A)` means that the action `A` was performed i.e., the action makes `A` true, whereas `do(\+A)` means that the action makes the atom `A` false.

12.3.2 Causal Exact Inference with `cplint`

When performing causal inference, evidence `E` may contain ground literals of the form `do(A)`, meaning that ground literal `A` is an action rather than an observation.

In this case, evidence `E` is partitioned into two conjunctions, `EO` containing only the observation atoms and `EA` containing all the literals `A` for which `E` contains `do(A)`. Let `remove_do` be the function taking as input a conjunction of `do` literals and returning `remove_do(EA)` $= \{$`A`$|$`do(A)` $\in$ `EA`$\}$.

The knowledge base is extended with

$$\$goal(\texttt{D})\ \texttt{:- PITA(G,D).}$$

as for non causal inference, plus

$$\$ev(\texttt{D})\ \texttt{:- PITA(EO,D).}$$

Then Algorithm 12.2 is used to obtain a new program on which conditional inference as in PITA is performed. The algorithm considers every action of the form `do(A)` $\in$ `EA` with `A = p(`t_1`,...,`t_n`)` or `A = \+p(`t_1`,...,`t_n`)` and, for each clause with `p(`u_1`,...,`u_n`,D)` in the head, it adds to the body the conjunction of constraints `dif(`u_1`,`t_1`),...,dif(`u_n`,`t_n`)`. Then the clause

$$\texttt{p(}t_1\texttt{,...,}t_n\texttt{,D) :- one(D).}$$

is added to the program for every action of the form `do(p(`t_1`,...,`t_n`))` (positive actions).

`dif/2` is a coroutine predicate that expresses disequality of terms. The actual test is delayed until the terms are sufficiently instantiated to be found different, or have become identical. The predicate is available in most Prolog systems and is usually implemented by means of attributed variables [157].

By using `dif/2`, the body of the clause fails as soon as a disequality is violated. If we had used the disunification predicate `\=/2`, we should have inserted the disequality constraints at the end of the body, just before the call to `get_var_n/4`, because at the beginning of the body some variables may not be instantiated. This would have resulted in a waste of computation, as failure would be obtained only after having resolved all the literals in the body. With `dif/2` failure may be obtained earlier.

The result is correct as shown by Theorem 12.1.

Algorithm 12.2 Algorithm for preparing the knowledge base for exact causal inference.

1: **function** PREPAREPITAKB(KB, `EA`) ▷ Program KB, set of literals appearing as *do* actions in the evidence `EA`
2: **for all** `do(A)` $\in$ `EA` with `A=p(`t_1`,...,`t_n`)` or `A=\+p(`t_1`,...,`t_n`)` **do**
3: **for all** clauses `C=p(`u_1`,...,`u_n`,D) :- B` **do**
4: Remove `C` from KB
5: Add `p(`u_1`,...,`u_n`,D) :- dif(`u_1`,`t_1`),...,dif(`u_n`,`t_n`),B` to KB
6: **end for**
7: **end for**
8: **for all** `do(A)` atom in `EA` with `A=p(`t_1`,...,`t_n`)` **do**
9: Add `p(`t_1`,...,`t_n`,D) :- one(D).` to KB
10: **end for**
11: **return** KB
12: **end function**

Theorem 12.1
Given a a goal `G` *and an evidence* `E`*, the probability for* `G` *to be true given that* `E` *holds* $P($`G`$|$`E`$)$ *on program* KB *has the same value as*

$$\text{PROB}(\text{PREPAREPITAKB}(KB, EA), G, EO).$$

Proof. By including the `dif/2` constraints in the body, we effectively make sure that, when evaluating the body of the clauses (causal laws), all groundings of the clauses whose head matches with one of the action atoms produce failure, resulting in the same effect as removing the ground causal law from the theory.

The addition of clauses

`p(`t_1`,...,`t_n`,D) :- one(D).`

for every positive action `do(p(`t_1`,...,`t_n`))` then ensures that `p(`t_1`,...,`t_n`)` is forced to true, and the absence of any clause for `p(`t_1`,...,`t_n`)` for negative actions `do(\+p(`t_1`,...,`t_n`))` ensures that `p(`t_1`,...,`t_n`)` is forced to false.

In this way we adopt the strategy of [48] for representing interventions in CP-logic. □

12.3.3 Causal Approximate Inference with `cplint`

As for PITA, the evidence `E` is partitioned into the conjunctions `EO` of observation atoms and `EA` of action atoms. Then the knowledge base is extended with

$$\$goal \texttt{ :- G.}$$

as for non causal inference, plus

$$\$ev \texttt{ :- EO.}$$

Then Algorithm 12.3, MCINTYRE's version of Algorithm 12.2, is used to preprocess the program before using MCINTYRE algorithms for conditional inference. You can notice that in Alg. 12.3 the variable `D` is missing (see predicates in Alg. 12.2), this is because variable `D` in exact inference is used to contain the BDD, but in approximate inference we just use sampling without building any BDDs. It is easy to see that

Algorithm 12.3 Algorithm for preparing the knowledge base for approximate causal inference.

1: **procedure** PREPAREMCKB(KB, `EA`) ▷ Program KB, set of literals appearing as *do* actions in the evidence `EA`
2: **for all** `do(A)` ∈ `EA` with `A=p(`t_1`,...,`t_n`)` or `A=\+p(`t_1`,...,`t_n`)` **do**
3: **for all** clauses `C=p(`u_1`,...,`u_n`) :- B` **do**
4: Remove `C` from KB
5: Add `p(`u_1`,...,`u_n`) :- dif(`u_1`,`t_1`),...,dif(`u_n`,`t_n`),B` to KB
6: **end for**
7: **end for**
8: **for all** `do(A)` atom in `EA` with `A=p(`t_1`,...,`t_n`)` **do**
9: Add `A` to KB
10: **end for**
11: **end procedure**

Theorem 12.1 holds also for MCINTYRE. Figure 12.3 shows the architecture of the cplint system with their module and algorithms used for causal inference.

12.3.4 Notable Examples

In this section we illustrate the implementation in `cplint` of two famous problems: the Simpson's paradox and the viral marketing problem.

12.3.5 Simpson's Paradox

The medicine study of Example 12.3.1 can be represented with the following LPAD[3].

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
:- action drug/0.
female:0.5.
recovery:0.6:- drug,\+ female.
recovery:0.7:- \+ drug,\+ female.
recovery:0.2:- drug,female.
recovery:0.3:- \+ drug,female.
drug:30/40:- \+ female.
```

[3]Also available at `http://http://cplint.ml.unife.it/example/inference/simpson.swinb`.

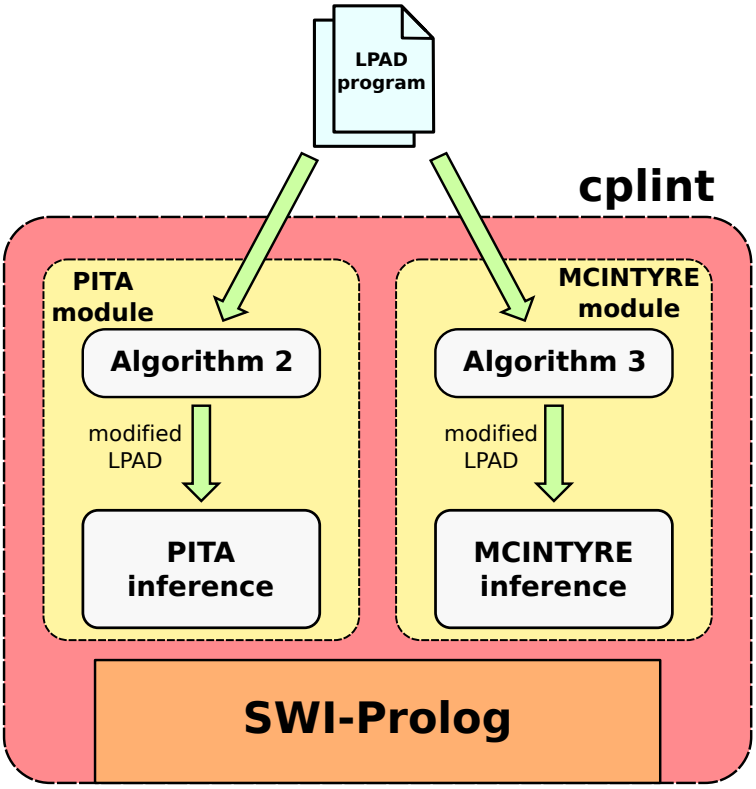


Figure 12.3: Architecture of cplint for causal inference.

```
drug:10/40:-female.
:-end_lpad.
```

Here, `:- action drug/0.` means that `drug/0` is a predicate that can be used to specify actions. We need the `:- action p/n` directive because the predicate `p` should be declared as *dynamic* in order to perform `retract/1` (execution of line 4 in Algorithm 12.2). In PITA the directive `:- action p/n` makes the predicate `p/n+2` *dynamic*. In MCINTYRE, instead, it has the effect to make `p/n` *dynamic*.

We query the conditional probabilities of recovery given treatment on the whole population and on the two subpopulations with:

```
?- prob(recovery,drug,P).
?- prob(recovery,\+ drug,P).
?- prob(recovery,(drug,female),P).
?- prob(recovery,(\+ drug,female),P).
?- prob(recovery,(drug,\+ female),P).
?- prob(recovery,(\+ drug,\+ female),P).
```

The results of these queries are those in the tables of Example 12.3.1.

If instead we want to know the probability of recovery given the action treatment (taking a drug), we must ask

```
?- prob(recovery,do(drug),P).
?- prob(recovery,do(\+ drug),P).
?- prob(recovery,(do(drug),female),P).
```

```
?- prob(recovery,(do(\+ drug),female),P).
?- prob(recovery,(do(drug),\+ female),P).
?- prob(recovery,(do(\+ drug),\+ female),P).
```

The results of the last four queries are the same as the last four conditional queries, so the probability of recovery in the two subpopulations is the same as that for the case of seeing rather than doing, as the observation of sex makes the arc from sex to drug irrelevant.

The results of the first two *do* queries instead differ from the conditional ones: they are respectively 0.4 and 0.5, showing that the drug is not beneficial and that the probability of recovery on the whole population is now in accordance with that in the subpopulations, in particular it is the weighted average of the probability of recovery in the subpopulations.

12.3.6 Viral Marketing

Let us now consider a viral marketing scenario inspired by [158]. A firm is interested in marketing a new product to its customers. These are connected in a social network that is known to the firm: the network represents the trust relationships between customers. The firm has decided to adopt a marketing strategy that involves giving the product for free to a number of its customers, in the hope that these influence the other customers and entice them to buy the product. The firm wants to choose the customers to which marketing is applied so that its return is maximised. This involves computing the probability that the non-marketed customers will acquire the product given the action to the marketed customers.

We can model this domain with an LPAD where the predicate `trust/2` encodes the links between customers in the social network and the predicate `has/1` is true for customers that possess the product, either received as a gift or bought. Predicate `trust/2` is defined by a number of certain facts, while predicate `has/1` is defined by two rules, one expressing the prior probability of a customer to buy the product and one expressing the fact that if a trusted customer has the product, then there is a certain probability that the trusting customer buys the product. The complete LPAD is shown in Figure 12.4[4]. The social network encoded by the program is represented in Figure 12.5. If the firm wants to estimate the effect of giving the product for free to customer 3 on the probability of customer 2 buying the product, the query to ask is

```
?- prob(has(2),do(has(3)),P).
```

This query on the program above returns 0.136. If instead we query

```
?- prob(has(2),has(3),P).
```

we get 0.407, showing that not distinguishing seeing from doing leads to an overly optimistic estimate.

12.3.7 Experiments

In this section we aim to evaluate the performance of causal reasoning with `cplint` while comparing exact inference, performed with PITA, with approximate inference,

[4] Also available at `http://cplint.ml.unife.it/example/inference/viral.swinb`.

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
:- action has/1.
has(_) : 0.1.
has(P) : 0.4 :- trusts(P, Q), has(Q).
trusts(2,1).
trusts(3,1).
trusts(3,2).
trusts(4,1).
trusts(4,3).
:- end_lpad.
```

Figure 12.4: LPAD for viral marketing.

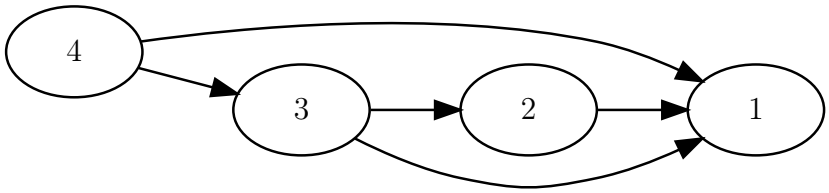


Figure 12.5: Social network for the viral marketing example.

performed with Metropolis-Hastings of MCINTYRE. Given the different focus of P-log, a comparison with this system would be unfair. Therefore we compare the performance of causal reasoning in `cplint` with regular probabilistic reasoning. All the experiments here presented were executed on a Linux machine equipped with a Intel Xeon E5-2630 v3 @ 2.40 GHz CPU with 8 GB of main memory.

In particular, we considered the viral marketing domain. We generated random social networks of increasing size and we evaluated random probabilistic and causal queries with an increasing number of evidence literals. The random social networks were generated as scale-free graphs according to the Barabasi-Albert model [159]. We used the `sample_pa`[5] function of the `igraph` R library to generate the graphs with parameter `m` set to 2 (the number of edges to be added at each time step is 2). We considered a number of nodes from 10 to 100 in steps of 10 and, for each number of nodes, we generated 10 graphs (for a total of 100 different generated graphs). For each number of nodes, we generated conjunctions of literals of the form `has(n)` where `n` is a node sampled uniformly at random from the set of nodes. For each number of literals from 2 to 8 in steps of 2 we generated 10 random conjunctions with that number of literals. For each conjunction $\mathtt{C}_l$ with l literals, we sampled uniformly a node `m` and we prepared the queries $P_l = P(\mathtt{has(m)}|\mathtt{C}_l)$ and $Q_l = P(\mathtt{has(m)}|\mathtt{do(C}_l\mathtt{)})$, where $\mathtt{do(C}_l\mathtt{)} = \{\mathtt{do(A)}|\mathtt{A} \in \mathtt{C}_l\}$.

Then we posed the queries P_l and Q_l to each of the 10 graphs for each number of nodes and we measured the execution time. The computed time was averaged over

[5] `http://igraph.org/r/doc/sample_pa.html`

the 10 graphs with the same number of nodes and the 10 conjunctions with the same number of literals. Hence we have 100 queries for each number of nodes. We set a timeout of 600 seconds for each query and we set to 1000 the number of samples for MCINTYRE.

The average runtime for conditional and causal queries are then plotted in Figures 12.6-12.9 as a function of the number of nodes. Tables 12.1-12.4 show the average timings with their 95% confidence intervals.

In particular, Figure 12.7 shows that with 4 evidence literals and a graph size larger than 60 nodes at least one conditional query with exact inference has encountered the timeout. Whereas causal queries (both with exact and approximate inference) and conditional queries with approximate inference are still feasible. Figures 12.8 and 12.9 show that at least one conditional query with exact inference has encountered the timeout for graphs with more than 10 nodes and queries with 6 evidence literals or more. In all the figures we can see that the running time of conditional inference increases with the size of the graphs, while the runtime of causal inference is roughly constant. In these experiments the average running time for causal approximate inference is below 130 milliseconds for every graph size, whereas causal exact inference is surprisingly faster than the approximate one and the average running time is below 4 milliseconds for every graph size. The causal exact inference is faster than the approximate one because, in our example, there is a small number of explanations for each causal query, therefore it takes less time to compute all the explanations than it does to sample the probabilistic logic program 1000 times. Table 12.5 reports the mean squared error of approximate causal inference (caus mcint). We can notice that the errors are less than $4 \cdot 10^{-3}$, proving that the proposed approximate approach gives results close enough to the exact ones.

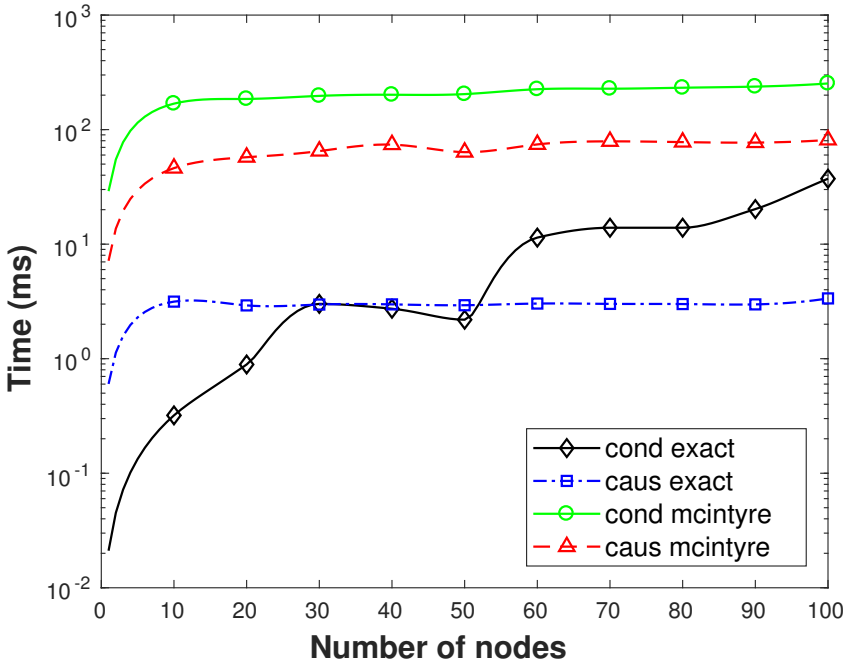


Figure 12.6: Average time for conditional and causal queries with 2 evidence literals.

Table 12.1: Execution time (in milliseconds) and 95% confidence intervals for conditional and causal queries with 2 evidence literals. The size of the datasets is expressed in number of nodes of the graph.

Inference method	**Size of the dataset**				
	10	**20**	**30**	**40**	**50**
cond exact	0.32 ± 0.09	0.89 ± 0.63	3.01 ± 1.59	2.73 ± 1.40	2.19 ± 1.52
caus exact	3.15 ± 0.55	2.92 ± 0.05	2.97 ± 0.05	2.98 ± 0.07	2.93 ± 0.06
cond mcint	168.62 ± 8.04	185.28 ± 6.15	197.05 ± 8.32	201.6 ± 5.91	204.22 ± 6.77
caus mcint	46.11 ± 3.19	57.42 ± 2.5	64.94 ± 4.08	73.96 ± 4.2	63.57 ± 3.44
Inference method	**Size of the dataset**				
	60	**70**	**80**	**90**	**100**
cond exact	11.41 ± 4.96	13.91 ± 7.31	13.91 ± 5.65	20.24 ± 8.34	37.29 ± 28.30
caus exact	3.03 ± 0.08	3.01 ± 0.10	3.00 ± 0.06	2.98 ± 0.09	3.36 ± 0.46
cond mcint	225.34 ± 8.68	227.66 ± 8.91	232.13 ± 10.47	237.46 ± 9.24	252.91 ± 12.9
caus mcint	74.27 ± 4.36	78.98 ± 6.6	77.81 ± 5.32	77.06 ± 6.77	81.26 ± 7.9

Table 12.2: Execution time (in milliseconds) and 95% confidence intervals for conditional and causal queries with 4 evidence literals. The dash means that the timeout was reached. The size of the datasets is expressed in number of nodes of the graph.

Inference method	**Size of the dataset**				
	10	**20**	**30**	**40**	**50**
cond exact	4.43 ± 1.27	66.98 ± 29.64	811.73 ± 446.72	936.18 ± 626.14	854.23 ± 682.3
caus exact	2.84 ± 0.07	2.95 ± 0.05	2.95 ± 0.07	3.04 ± 0.06	2.9 ± 0.06
cond mcint	236.18 ± 8.89	289.33 ± 11.77	350.97 ± 20.46	381.16 ± 25.24	364.11 ± 21.96
caus mcint	41.91 ± 3.25	56.88 ± 4.14	65.93 ± 4.48	79.38 ± 4.48	63.48 ± 3.38
Inference method	**Size of the dataset**				
	60	**70**	**80**	**90**	**100**
cond exact	2372.11 ± 1291.21	–	–	–	–
caus exact	3.04 ± 0.08	3.06 ± 0.08	3.03 ± 0.07	3.07 ± 0.06	3.37 ± 0.38
cond mcint	438.48 ± 36.96	453.89 ± 34.03	464.05 ± 35.7	482.93 ± 32.17	522.62 ± 39.74
caus mcint	71.89 ± 5.27	82.19 ± 6.04	78.64 ± 5.98	94.64 ± 7.01	103.73 ± 10.01

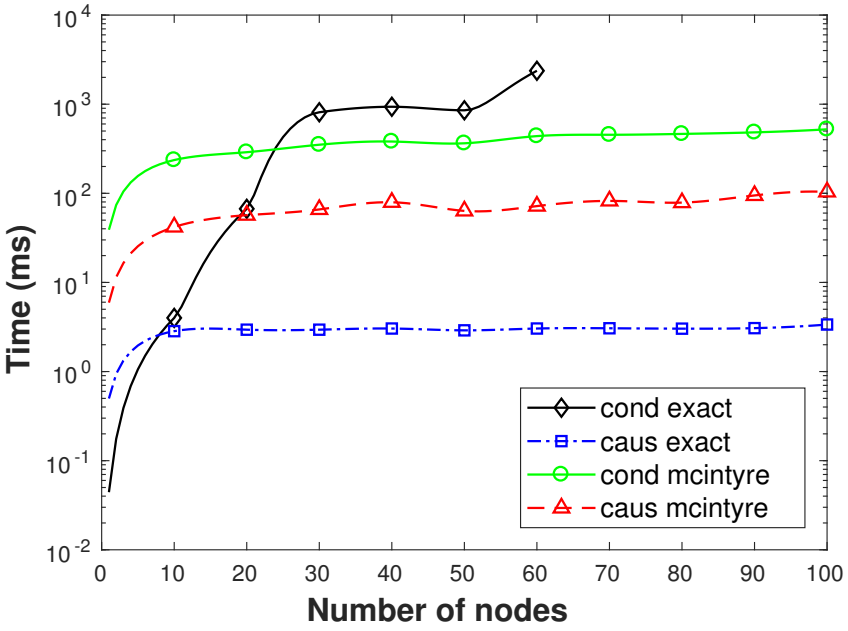


Figure 12.7: Average time for conditional and causal queries with 4 evidence literals.

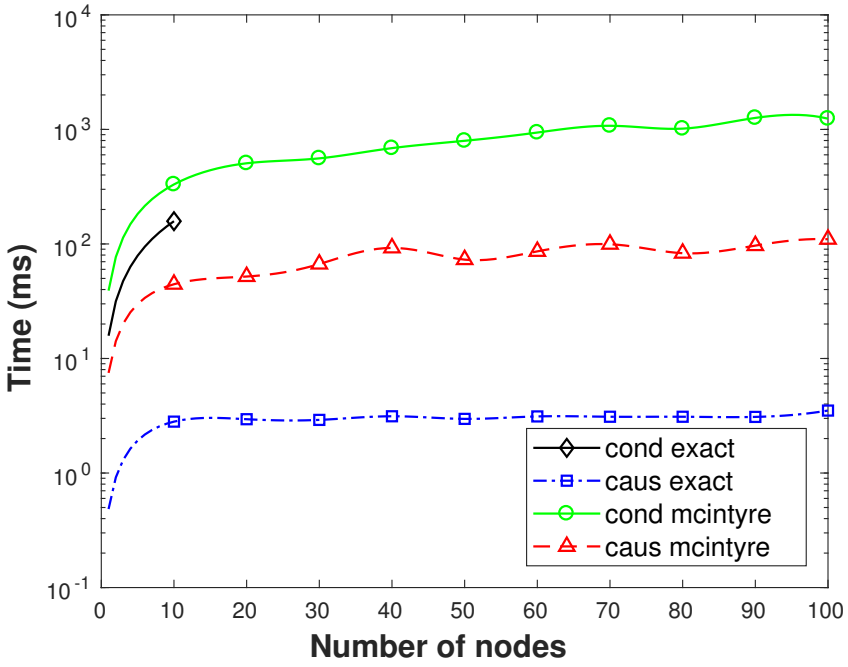


Figure 12.8: Average time for conditional and causal queries with 6 evidence literals.

Table 12.3: Execution time (in milliseconds) and 95% confidence intervals for conditional and causal queries with 6 evidence literals. The dash means that the timeout was reached. The size of the datasets is expressed in number of nodes of the graph.

Inference method	**Size of the dataset**				
	10	**20**	**30**	**40**	**50**
cond exact	158.37 ± 38.36	–	–	–	–
caus exact	2.81 ± 0.08	2.95 ± 0.06	2.91 ± 0.06	3.13 ± 0.08	2.97 ± 0.07
cond mcint	331 ± 14.59	506.29 ± 40.93	558.82 ± 68.84	686.51 ± 91.9	795.54 ± 122.04
caus mcint	44.5 ± 1.4	51.95 ± 4.5	66.74 ± 5.25	92.36 ± 7.05	72.96 ± 4.31
Inference method	**Size of the dataset**				
	60	**70**	**80**	**90**	**100**
cond exact	–	–	–	–	–
caus exact	3.12 ± 0.08	3.1 ± 0.08	3.1 ± 0.07	3.09 ± 0.07	3.51 ± 0.54
cond mcint	939.05 ± 248.19	1075.76 ± 149.54	1015.65 ± 160.3	1260.15 ± 199.99	1240.61 ± 208.2
caus mcint	86.13 ± 6.21	99.42 ± 7.7	83.02 ± 7.53	96.71 ± 8.15	109.95 ± 12.87

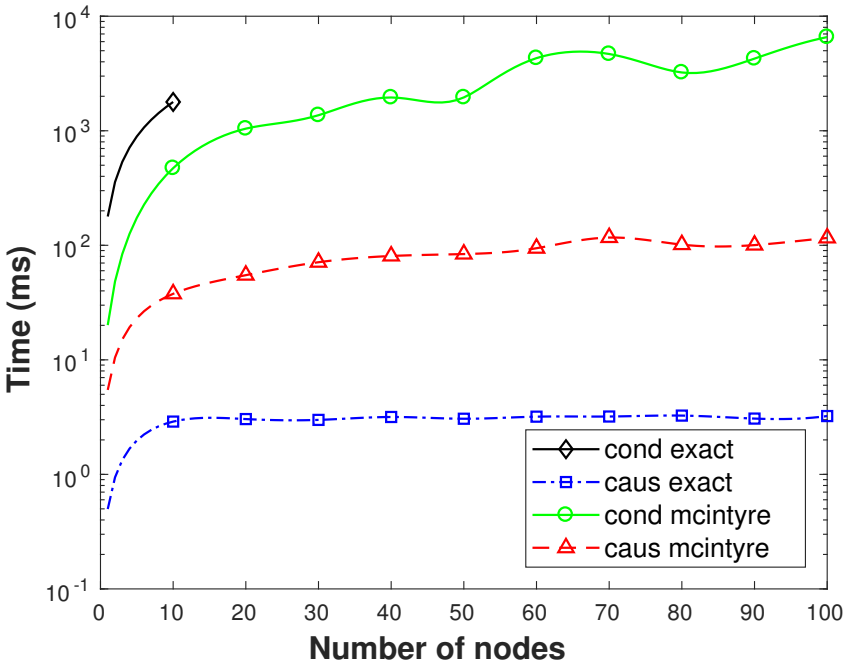


Figure 12.9: Average time for conditional and causal queries with 8 evidence literals.

Table 12.4: Execution time (in milliseconds) and 95% confidence intervals for conditional and causal queries with 8 evidence literals. The dash means that the timeout was reached. The size of the datasets is expressed in number of nodes of the graph.

Inference method	**Size of the dataset**				
	10	**20**	**30**	**40**	**50**
cond exact	1784.48 ± 451.27	–	–	–	–
caus exact	2.89 ± 0.07	3.04 ± 0.06	2.99 ± 0.07	3.17 ± 0.08	3.06 ± 0.07
cond mcint	471.56 ± 34.17	1043.48 ± 255.12	1366.8 ± 232.7	1952.12 ± 382.1	1954.56 ± 336.25
caus mcint	37.71 ± 2.28	54.86 ± 3.42	71.15 ± 4.29	80.67 ± 8.1	83.86 ± 5.85
Inference method	**Size of the dataset**				
	60	**70**	**80**	**90**	**100**
cond exact	–	–	–	–	–
caus exact	3.19 ± 0.08	3.2 ± 0.08	3.26 ± 0.09	3.07 ± 0.06	3.23 ± 0.09
cond mcint	4306.83 ± 1759.19	4679.31 ± 1304.03	3227.32 ± 1234.65	4265.53 ± 1375.54	6576.63 ± 2313.33
caus mcint	94.28 ± 7.36	116.85 ± 9.25	101.36 ± 9.51	100.42 ± 7.34	115.94 ± 9.33

Table 12.5: Mean Squared Error for approximate causal inference (caus mcintyre). All the values must be multiplied by 10^{-3}. The size of the datasets is expressed in number of nodes of the graph.

Evidence literals	**Size of the dataset**									
	10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
2	2.5	1.7	2.4	2.4	1.6	2.3	2.3	3.0	2.0	2.6
4	0.9	1.8	2.4	2.7	1.6	2.6	2.6	2.2	2.7	2.9
6	1.2	1.4	2.5	3.9	1.7	2.7	2.3	2.3	2.1	2.3
8	0.9	2.2	1.5	3.2	2.3	2.3	3.1	2.0	2.5	2.0

12.4 Hybrid Probabilistic Logic Programs with `cplint`

Up to now we have considered only discrete random variables and discrete probability distributions. How can we consider continuous random variables and probability density functions, for example real variables following a Gaussian distribution?

`cplint` allows the specification of density functions over arguments of atoms in the head of rules. To specify a probability density on an argument `Var` of an atom `A` you can use rules of the form

```
A : Density :- Body.
```

where `Density` is a special atom identifying a probability density on variable `Var` and `Body` (optional) is a regular clause body. Allowed `Density` atoms are:

- `uniform(Var,L,U)`: `Var` is uniformly distributed in $[L, U]$.
- `gaussian(Var,Mean,Variance)`: `Var` follows a Gaussian distribution with mean `Mean` and variance `Variance`.
- `dirichlet(Var,Par)`: `Var` is a list of real numbers following a Dirichlet distribution with parameters α specified by the list `Par`.
- `gamma(Var,Shape,Scale)`: `Var` follows a gamma distribution with shape parameter `Shape` and scale parameter `Scale`.
- `beta(Var,Alpha,Beta)`: `Var` follows a beta distribution with parameters `Alpha` and `Beta`.

This syntax can be used to describe also discrete distribution, with either a finite or countably infinite support:

- `discrete(Var,D)` or `finite(Var,D)`: `A` is an atom containing variable `Var` and `D` is a list of couples `Value:Prob` assigning probability `Prob` to `Value`
- `uniform(Var,D)`: `A` is an atom containing variable `Var` and `D` is a list of values each taking the same probability (1 over the length of `D`).
- `poisson(Var,Lambda)`: `Var` follows a Poisson distribution with parameter `Lambda`.

This type of clauses are called Distributional Clauses [160].

Example 12.4.1
Consider the following LPAD rule

```
g(X,Y) : gaussian(Y,0,1) :- object(X).
```

`X` *takes terms while* `Y` *takes real numbers as values. The clause states that, for each* `X` *such that* `object(X)` *is true, the values of* `Y` *such that* `g(X,Y)` *is true follow a Gaussian distribution with mean 0 and variance 1. You can think of an atom such as* `g(a,Y)` *as an encoding of a continuous random variable associated with term* `g(a)`.

This kind of probabilistic logic programs where some of the random variables are continuous are called Hybrid Probabilistic Logic Programs.

If an atom encodes a continuous random variable (such as `g(X,Y)` in Example 12.4.1), asking the probability that a ground instantiation, such as `g(a,0.3)`, is true is not meaningful, as the probability that a continuous random variables takes a specific value is always 0. In this case you are more interested in computing the distribution of `Y` of a goal `g(a,Y)`, possibly after having observed some evidence. If the evidence is on an atom defining another continuous random variable, the definition of conditional probability cannot be applied, as the probability of the evidence would be 0 and so the fraction would be undefined. This problem is resolved in [161] by providing a definition using limits.

12.4.1 Sampling the Arguments of Unconditional Queries over Hybrid Programs

If the query is unconditional, we can use approximate inference with Monte Carlo sampling as described in the Subsection 12.2.2. When we have continuous random variables, we are interested in sampling arguments of goals representing continuous random variables. In this way we can build a probability density of the sampled argument. To do that it is possible to use the predicate

```
mc_sample_arg(:Query:atom,+Samples:int,?Arg:var, -Values:list).
```

that returns in `Values` a list of couples `L-N` where `L` is the list of values of `Arg` for which `Query` succeeds in a world sampled at random and `N` is the number of samples returning that list of values.

Example 12.4.2 Gaussian mixture
As example, let us consider the following program that models the mixture of two Gaussians.

```
heads : 0.6; tails : 0.4.
g(X) : gaussian(X,0,1).
h(X) : gaussian(X,5,2).
mix(X) :- heads, g(X).
mix(X) :- tails, h(X).
```

A biased coin is thrown, if it lands heads, X in mix(X) is sampled from a Gaussian with mean 0 and variance 1. If it lands tails, X is sampled from a Gaussian with mean 5 and variance 2.

We can now perform the query

```
mc_sample_arg(mix(X),1000,X,Values).
```

`Values` *will contain a list of couples* `L-N` *where* `L` *is the list of values of* `X` *for which query succeeds in a world sampled at random and* `N` *is the number of samples returning* `L`*. Notice that, in every couple* `L-N`*,* `L` *will contain just one element and* `N` *will be always 1. This is because the random variable* `X` *is continuous and* `mix(X)` *always succeeds exactly once, therefore the predicate* `mc_sample_arg/4` *will sample 1000 different worlds and every world will have a different value for* `X`*.*

12.4.2 Conditional Queries over Hybrid Logic Programs

As in the previous subsection we are interested in sampling arguments of goals representing continuous random variables (CRVs), but this time we have also some atoms as evidence.

To perform this kind of query we must distinguish three cases depending on what type of evidence we have:

- The evidence does not contain atoms with CRVs (the probability of evidence is not 0).
- The evidence contains non-ground atoms with CRVs, (the probability of evidence is not 0).
- The evidence contains groundings of atoms with CRVs (its probability is 0).

For the first two cases you can use rejection sampling or Metropolis-Hastings. However, when evidence on ground atoms have continuous values as arguments, we cannot use rejection sampling or Metropolis-Hastings, as the probability of the evidence is 0, but we can use **likelihood weighting** [161] or **particle filtering** [162, 163] to obtain samples of the continuous arguments of a goal (see Subsubsection 12.4.2.3).

12.4.2.1 Case 1: evidence on atoms without CRVs

To sample the arguments of the queries with rejection sampling and Metropolis-Hastings MCMC, we can use the following predicates

```
mc_rejection_sample_arg(:Query:atom,:Evidence:atom, +Samples:int,
  ?Arg:var,-Values:list).
mc_mh_sample_arg(:Query:atom,:Evidence:atom, +Samples:int,
  +Lag:int,?Arg:var,-Values:list).
```

Example 12.4.3
Let us consider the same program of Example 12.4.2. We want to take 1000 samples of `X` *in* `mix(X)` *given that* `heads` *was true using rejection sampling and Metropolis-Hastings MCMC. We can do it with the following predicates*

```
?- mc_rejection_sample_arg(mix(X),heads,1000,X,Values).
?- mc_mh_sample_arg(mix(X),heads,1000,2,X,Values).
```

12.4.2.2 Case 2: evidence on non-ground atoms with CRVs

We discuss this case by means of the example below.

Example 12.4.4
Let us consider the same program of Example 12.4.2. We want to take 1000 samples of `X` *in* `mix(X)` *given that* `X > 2` *was true using rejection sampling and Metropolis-Hastings MCMC. We can do that with the following queries*

```
?- mc_rejection_sample_arg(mix(X),(mix(Y),Y>2),1000,X,Values).
?- mc_mh_sample_arg(mix(X),(mix(Y),Y>2),1000,2,X,Values).
```

12.4.2.3 Case 3: evidence on groundings of atoms with CRVs

When we have evidence on ground atoms that have continuous values as arguments, we need to use **likelihood weighting** [161] or **particle filtering** [162, 163] to obtain samples of the continuous arguments of a goal.

Likelihood Weighting For each sample to be taken, likelihood weighting uses a meta-interpreter to find a sample where the goal is true, randomizing the choice of clauses when more than one resolves with the goal, in order to obtain an unbiased sample. This meta-interpreter is similar to the one used to generate the first sample in Metropolis-Hastings.

Then a different meta-interpreter is used to evaluate the weight of the sample. This meta-interpreter starts with the evidence as the query and a weight of 1. Each time the meta-interpreter encounters a probabilistic choice over a continuous variable, it first checks whether a value has already been sampled. If so, it computes the probability density of the sampled value and multiplies the weight by it. If the value had not been sampled, it takes a sample and records it, leaving the weight unchanged. In this way, each sample in the result has a weight that is 1 for the prior distribution and that may be different from the posterior distribution, reflecting the influence of evidence.

The predicate

```
mc_lw_sample_arg(:Query:atom,:Evidence:atom,+N:int,?Arg:var,
   -ValList:list).
```

returns in `ValList` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds and `W` is the weight computed by likelihood weighting according to `Evidence` (a conjunction of atoms is allowed here).

Example 12.4.5 Bayesian estimation
Consider the following LPAD[6] *based on a problem proposed on the Anglican [164] web site*[7]*.*

```
value(I,X) :-
    mean(M),
    value(I,M,X).
mean(M): gaussian(M,1.0,5.0).
value(_,M,X): gaussian(X,M,2.0).
```

We are trying to estimate the true value of a Gaussian distributed random variable, given some observed data. The variance is known to be 2 and we suppose that the mean has a Gaussian distribution with mean 1 and variance 5.

Suppose we have taken different measurement (e.g. at different times), indexed with an integer. Given that we observe 9 and 8 at indexes 1 and 2, we can ask how the distribution of the random variable (value at index 0) changes, with the query

```
?- mc_lw_sample_arg(value(0,X),(value(1,9),value(2,8)),10000,X,V).
```

[6] `http://cplint.ml.unife.it/example/inference/gauss_mean_est.pl`

[7] `http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=gaussian-posteriors`

This query takes $10,000$ *samples of the argument* `X` *of* `value(0,X)` *before and after the observation of* `value(1,9),value(2,8)`.

Example 12.4.6 Kalman filter
The following LPAD[8] *(adapted from [154]) encodes a Kalman filter, i.e., a Hidden Markov model with a real value as state and a real value as output.*

```
kf(N,O,T) :- init(S), kf_part(0,N,S,O,T).
kf_part(I,N,S,[V|RO],T) :-
    I < N, NextI is I+1,
    trans(S,I,NextS),emit(NextS,I,V),
    kf_part(NextI,N,NextS,RO,T).
kf_part(N,N,S,[],S).
trans(S,I,NextS) :-
    {NextS =:= E+S},
    trans_err(I,E).
emit(NextS,I,V) :-
    {V =:= NextS+X},
    obs_err(I,X).
init(S) : gaussian(S,0,1).
trans_err(_,E) : gaussian(E,0,2).
obs_err(_,E) : gaussian(E,0,1).
```

The next state is given by the current state plus Gaussian noise (with mean 0 and variance 2 in this example) and the output is given by the current state plus Gaussian noise (with mean 0 and variance 1 in this example). A Kalman filter can be considered as modeling a random walk of a single continuous state variable with noisy observations.

Continuous random variables are involved in arithmetic expressions (in the predicates `trans/3` *and* `emit/3`*). It is often convenient, as in this case, to use CLP(R) constraints so that the same clauses can be used both to sample and to evaluate the weight of the sample on the basis of the evidence, otherwise different clauses have to be written.*

Given that at time 0 the value 2.5 was observed, what is the distribution of the state at time 1 (filtering problem)? Likelihood weighting can be used to condition the distribution on evidence on a continuous random variable (evidence with probability 0). CLP(R) constraints allow both sampling and weighting samples with the same program: when sampling, the constraint `{V=:=NextS+X}` *is used to compute* `V` *from* `X` *and* `NextS`*. When weighting, the constraint is used to compute* `X` *from* `V` *and* `NextS`*. The above query can be expressed in* `cplint`*, e.g. with 10000, as follows*

```
?- mc_lw_sample_arg(kf(1,_O2,T),kf(1,[2.5],_T),10000,T,L).
```

Particle filtering When you have a dynamic model and observations on continuous variables for a number of time points, or your evidence is represented by many atoms, likelihood weighting has numerical stability problems, as samples' weight goes rapidly

[8] `http://cplint.ml.unife.it/example/inference/kalman_filter.pl`

to 0. In this case, particle filtering can be useful, because it periodically resamples the individual samples/particles so that their weight is reset to 1.

In particle filtering, the evidence is a list of atoms. Each sample is weighted by the likelihood of an element of the evidence and constitutes a particle. After weighting, particles are resampled and the next element of the evidence is considered.

The predicate

```
mc_particle_sample_arg(:Query:atom, +Evidence:term, +Samples:int,
    ?Arg:var,-Values:list).
```

samples the argument `Arg` of `Query` using particle filtering given that `Evidence` is true. `Evidence` is a list of goals and `Query` can be either a single goal or a list of goals.

When `Query` is a single goal, the predicate returns in `Values` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds in a particle in the last set of particles, and `W` is the weight of the particle. For each element of `Evidence`, the particles are obtained by sampling `Query` in each current particle and weighting the particle by the likelihood of the evidence element.

When `Query` is a list of goals, `Arg` is a list of variables, one for each query of `Query`; in this case `Arg` and `Query` must have the same length as `Evidence`. `Values` is then a list of the same length as `Evidence` and each of its elements is a list of couples `V-W` where `V` is a value of the corresponding element of `Arg` for which the corresponding element of `Query` succeeds in a particle, and `W` is the weight of the particle. For each element of `Evidence`, the particles are obtained by sampling the corresponding element of `Query` in each current particle and weighting the particle by the likelihood of the evidence element.

Example 12.4.7
Consider the LPAD and the conditional query in Example 12.4.5. We can ask the same thing by using particle filtering, with the query

```
?- mc_particle_sample_arg(value(0,X),[value(1,9),value(2,8)],
    100000,X,V).
```

12.5 cplint on SWISH: a Web interface for cplint

`cplint` on SWISH is a web application that allows users to perform reasoning tasks on probabilistic logic programs. It uses the reasoning algorithms included in the `cplint` suite, including exact and approximate inference and parameter and structure learning.

12.5.1 SWISH

SWISH[9] is a web application that allows the user to write Prolog programs and ask queries through the browser. SWISH was originally written by Torbjörn Lager and later extended by Jan Wielemaker. SWISH is based on SWI-Prolog and uses its Pengines library [165], which allows to create Prolog engines from an ordinary Prolog thread, from another Pengine, or from JavaScript running in a web client.

[9] `http://swish.swi-prolog.org/`

The SWISH page is divided into three panes, one with a program editor (on the left), one with a query editor (on the bottom right) and one that shows the query results (on the top right). When the user hits return after writing a query, a *runner* is created that collects the text in the program editor (if any) and the query and sends them to the server, which creates a Pengine (Prolog Engine). The Pengine compiles the program into a temporary private module. The Pengine assesses whether executing the query can compromise the system. If this fails, an error is displayed. If the query is considered safe, it executes the query and communicates with the runner about the results using JSON messages.

A Pengine is composed of a Prolog thread, a dynamic clause database (private to the Pengine), a message queue for incoming requests and a message queue for outgoing responses.

Pengines follow a master/slave architecture in which the master creates a Pengine on the slave and posts a query to it. The conversations between the master and the slave follow a communication protocol called the Prolog Transport Protocol (PLTP) that is layered on top of HTTP.

We now show an example from [165]: we use `pengine_create/1` to create a slave Pengine in a remote Pengine server.

```
:- use_module(library(pengines)).
main :-
    pengine_create([
        server('http://pengines.org'),
        src_text("
            q(X) :- p(X).
            p(a). p(b). p(c).
        ")
    ]),
    pengine_event_loop(handle, []).

handle(create(ID, _)) :-
    pengine_ask(ID, q(X), []).
handle(success(ID, [X], false)) :-
    writeln(X).
handle(success(ID, [X], true)) :-
    writeln(X),
    pengine_next(ID, []).
```

The option `src_text` is used to send the program to be queried in textual form to the Pengine. `pengine_event_loop/2` is used to start an event loop that listens for event terms and calls `handle/1` on them. If the event term is `create(ID,_)`, it means that the Pengine with id `ID` has been created and the event handler uses `pengine_ask/3` to ask the query. Predicate `pengine_ask/3` is deterministic, the results of the query will be returned in the form of event terms. If the event term is of the form `success(ID, Query, More)`, `ID` is the Pengine's id that succeeded in solving the query, `Query` holds an instantiation of the query and `More` is either `true` or `false`, indicating whether we can expect the Pengine to be able to return more solutions or not. If `More` is true, `handle/1` calls `pengine_next/2` to get the following solution. Thus running `main/0`

will write the terms `q(a)`, `q(b)` and `q(c)` to standard output.

Code sent to Pengines is executed in a "sandboxed" environment that ensures that only predicates that do not have side effects, such as accessing the file system, loading foreign extensions, defining other predicates outside the sandbox environment, etc., are called. Goals' safety is validated using a call to `safe_goal/1` of `library(sandbox)` prior to execution.

SWI-Prolog also offers a JavaScript library `pengine.js` that allows the creation of Pengine JavaScript objects. These, in turn, create Pengine objects on the server that can be queried from JavaScript.

The SWISH web server is implemented by the SWI-Prolog HTTP package, a series of libraries for serving data on HTTP [166].

SWISH exploits TogetherJS[10] in order to make the development of the code collaborative. TogetherJS is an open source JavaScript library built and hosted by Mozilla. This library permits a real time interaction between users and offers different built-in features:

Audio and Text Chat The collaborators can chat by talking or texting to each other.

User Focus The collaborators see each other's mouse cursors and clicks.

Co-browsing The collaborators can follow each other to different pages on the same domain.

Real time content sync The content is synchronized between all the collaborators.

It is possible to start collaborating on SWISH by clicking the item "File" in the menu bar and then clicking on "Collaborate..". The TogetherJS dock will appear and you can invite another user by sharing the generated link.

12.5.2 `cplint` on SWISH

In order to implement `cplint` on SWISH, we had to modify the foreign language C library that PITA uses as interface to CUDD so that different threads can use it at the same time. In fact, the library makes use of static global variables that hold data structures including the BDD manager and the association between the random variables and CUDD variables. If two different threads use the library, there would be a conflict on these variables. Therefore, we added an extra argument `Environment`, shortened `Env`, to all the library predicates defined in Subsection 12.2.1. This argument holds data structures regarding an individual query, including the BDD manager, and allows multiple threads to compute the probability of different queries, one thread per query.

So `init(-Env)`, when called, returns a pointer to a data structure storing the environment that must be given as input to all the other predicates:

- `zero(+Env,-D), one(+Env,-D)`

[10] `https://togetherjs.com/`

- `and(+Env,+D1,+D2,-D0), or(+Env,+D1,+D2,-D0), not(+Env,+D1,-D0)`
- `equality(+Env,+Var,+Value,-D)`
- `ret_prob(+Env,+D,-P)`
- `end(+Env)`

As a consequence, the PITA transformation of the LPAD must receive the variable `Env` that stores the environment.

Note that the `init/1` predicate, which initializes the BDD manager, is called for each query, thus each query, and so each Binary Decision Diagram, is handled by a different BDD manager. In this way if a thread crashes, the other ones will not be affected, the client will be notified of the failure and a new query can be immediately started.

To allow Pengines to execute the PITA library predicates, these must be declared safe by the code:

```
:- multifile sandbox:safe_primitive/1.

sandbox:safe_primitive(pita:init).
sandbox:safe_primitive(pita:ret_prob(_,_)).
...
```

in the `pita` module file.

The PITA library was also modified with respect to the application of the transformation of the program. While PITA uses a predicate `load/1` that loads the program file and applies the transformation to it, we decided to use term expansion through the predicate `term_expansion/2`, a de-facto standard in Prolog for source-to-source transformations. When compiling a module, SWI-Prolog will consider each term `T` in the program one by one and apply `term_expansion(T,NewT)`, then it will compile `NewT` instead of `T`. So if the user provides clauses for the `term_expansion/2` predicate, the system will compile a modified version of the input.

After loading `pita` with `use_module(library(pita))`, the PITA predicate `set/2` must be used to set the PITA flag `compiling` to `on`. All the clauses for `term_expansion/2` check this flag before performing the transformation. If it is not set to `on`, the transformation is not applied. After setting `compiling` to `on`, a file containing an LPAD is consulted to be translated into Prolog and loaded in memory.

Similarly, if a file containing an LPAD includes the directives `:-use_module(library(pita)).` and `:-set(compiling,on).` at the beginning, when it is consulted from the top level it is transformed and loaded in memory.

`cplint` on SWISH has the interface shown in Figure 12.10.

It allows the user to write an LPAD in the left pane and write a query in the bottom right pane. When the user presses enter at the end of the query or presses the Run! button, a Pengine is created with the program. This is done by the `runner.js` JavaScript file that creates a new Pengine object. The creation of the object was modified by adding to the program source some directives for loading the `pita` library, for disabling the check for discontiguous clauses and for enabling compilation. This is done by the following snippet of `runner.js`:

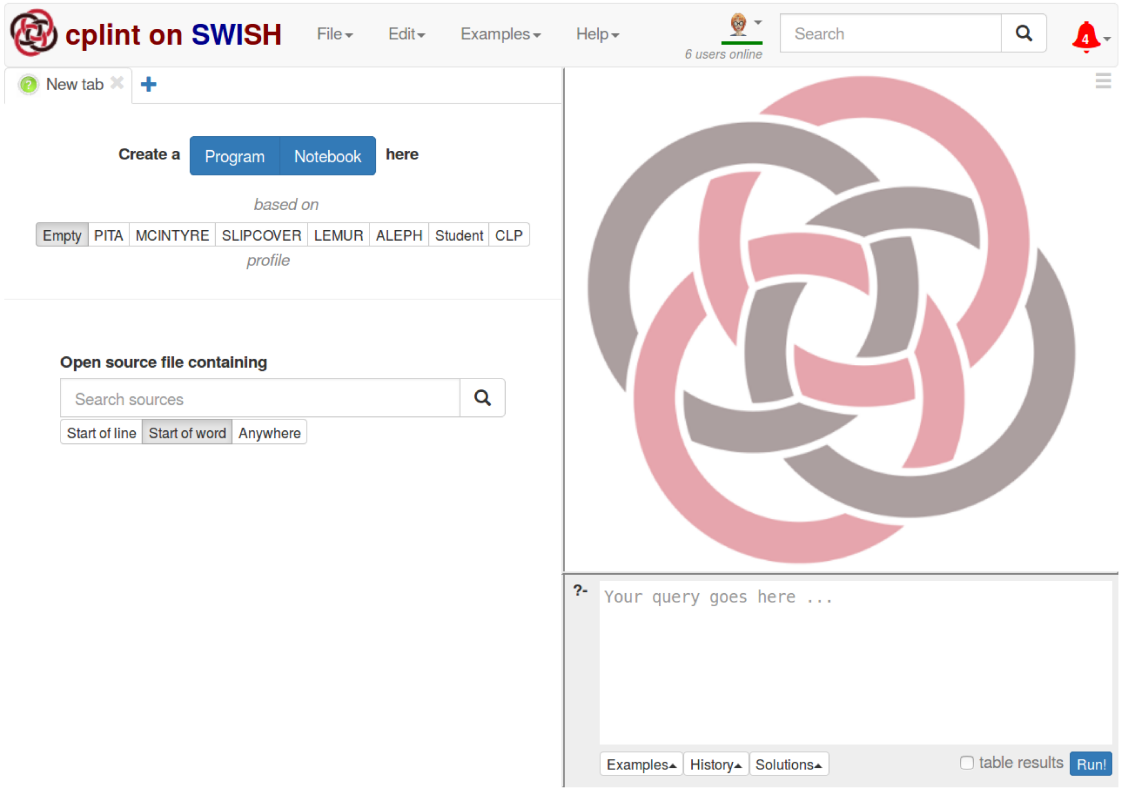


Figure 12.10: `cplint` on SWISH interface.

```
data.prolog = new Pengine({
  ...
  src: ":-use_module(library(pita)).
    :-style_check(-discontiguous).
    :-set(compiling,on). "
    + query.source,
  ...
  oncreate: handleCreate,
  ...
  });
```

that stores a new Pengine object in the runner's `data.prolog` attribute. `query.source` holds the program text.

The `handleCreate` function is performed at the creation of the Pengine and was modified to allow the computation of the probability. The query given by the user is sent to the Pengine with the `ask` method in a transformed form, i.e. the query atom without the full stop is inserted into a call to `s/2` in this way:

```
function handleCreate() {
  var elem = this.pengine.options.runner;
  var data = elem.data('prologRunner');
  this.pengine.ask("s(" +
          termNoFullStop(data.query.query)
             +",Prob)");
```

```
        elem.prologRunner('setState', "running");
    }
```

Here `data.query.query` is a string containing the query. The top right pane will then show the value of the `Prob` variable, together with the other variables' values possibly present in the query.

`cplint` on SWISH is robust as each query is executed in a separate thread with a time limit of 60 seconds. The server, in case of a problem, simply kills the thread that raised the error and returns the corresponding error message to the client. We ran several stress tests submitting queries that took more that 60 seconds and saturated the available memory before the time-out: the corresponding threads were simply killed at the time-out or at the memory saturation without affecting the other threads. However, it should be noted that `cplint` on SWISH is not appropriate for heavy computations, for which an execution on a local installation is better. `cplint` on SWISH was designed for developing and experimenting with the system, also in a collaborative way. The time limit for the execution of the queries was chosen to ensure the responsiveness of the server to other clients' requests. Our tests showed that the system remains active even in the presence of a high load.

The system contains a wide variety of examples, representing many probabilistic models such as Markov Logic Networks, generative models, Gaussian processes, Gaussian mixtures, Dirichlet processes, Bayesian estimation and Kalman filters. As such, the system shows the soundness of PLP also from a software engineering point of view, opening the way to complex industrial/real world applications. Moreover, a complete online tutorial [9] is available at `http://ds.ing.unife.it/~gcota/plptutorial/`.

12.5.3 Examples

In this subsection we show some inference examples by using `cplint` on SWISH. Further examples for can be found in [167] and online at `http://cplint.ml.unife.it/`.

Example 12.5.1
Let us consider the LPAD in Example 6.2.1. In order to calculate the probability that a pandemic arises, you can call the query:

```
?- prob(pandemic,Prob).
```

or

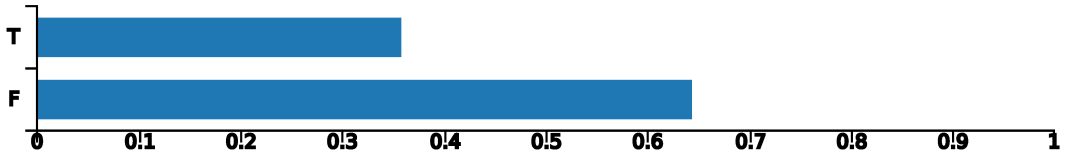
```
?- prob_bar(pandemic,Prob).
```

The latter shows the probabilistic results of the query as a histogram (Figure 12.11a). The corresponding BDD can be obtained with:

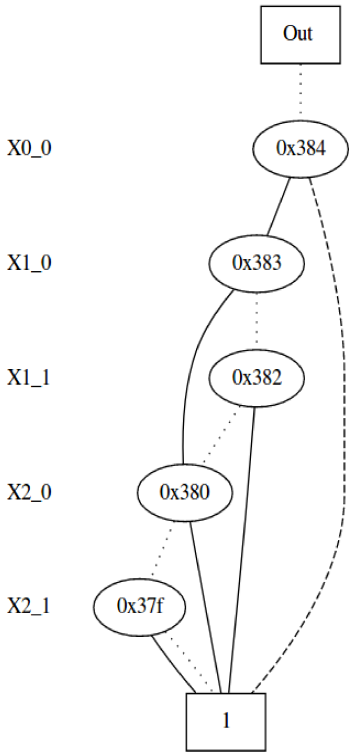
```
?- bdd_dot_string(pandemic,BDD,Var).
```

and is represented in Figure 12.11. A solid edge indicates a 1-child, a dashed edge indicates a 0-child and a dotted edge indicates a negated 0-child. Each level of the BDD is associated with a variable of the form X_{IJ} indicated on the left: I indicates the multivalued variable index and J the index of the Boolean variable of I. The table `Var`

contains the associations between the rule groundings and the multivalued variables: the first column contains the multivalued variable index, the second column contains the rule index, corresponding to its position in the program, and the last column contains the list of constants grounding the rule, each replacing a variable in the order of appearance in the rule.



(a) Histogram of the probabilistic result of query `pandemic` in Example 12.5.1.



(b) BDD for query `pandemic` in Example 12.5.1.

Figure 12.11: Graphical representations for query `pandemic` in Example 12.5.1.

Example 12.5.2 Gaussian mixture cont.
Consider the LPAD and the query performed in Example 12.4.2. With the following query

```
?- mc_sample_arg(mix(X),10000,X,L0), histogram(L0,40,Chart).
```

we can plot the histogram of the probability density function of the argument `X` *of* `mix(X)`*, shown in Figure 12.12.*

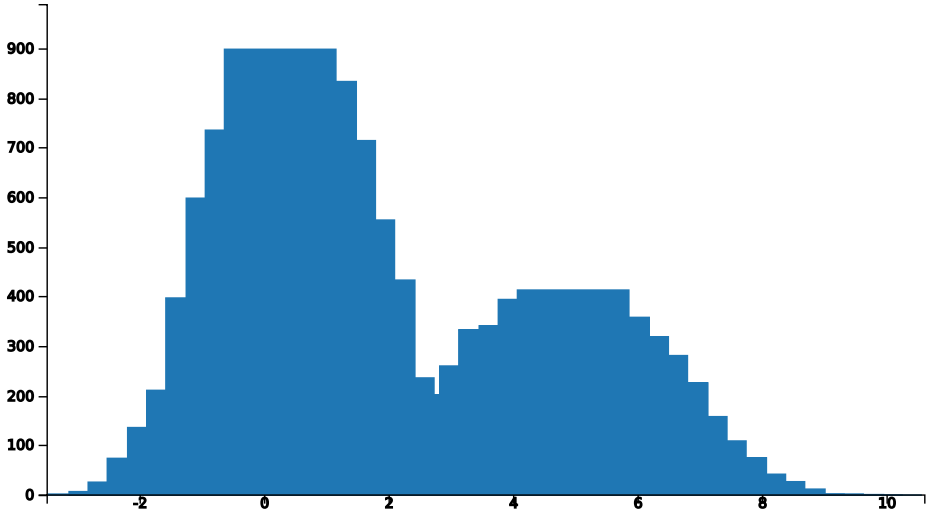


Figure 12.12: Density of `X` of `mix(X)` obtained by the query in Example 12.5.2.

Example 12.5.3 Bayesian estimation cont.
Consider the LPAD and the query performed in Example 12.4.5 and the query in Example 12.4.7. The query

```
?- mc_sample_arg(value(0,Y),10000,Y,L0),
   mc_lw_sample_arg(value(0,X),(value(1,9), value(2,8)),10000,X,L),
   densities(L0,L,40,Chart).
```

takes 10,000 samples of the argument `X` *of* `value(0,X)` *before and after the observation of* `value(1,9),value(2,8)` *and draws the prior and posterior densities of the samples using a line chart. Figure 12.13a shows the resulting graph where the posterior is clearly peaked at around 7.*

We can obtain similar results by using particle filtering, but have to use `mc_particle_sample_arg/5` *instead of* `mc_lw_sample_arg/5`*. The query*

```
?- mc_sample_arg(value(0,Y),Samples,Y,L0),
   mc_particle_sample_arg(value(0,X),[value(1,9),value(2,8)],
       Samples,X,L),
   densities(L0,L,NBins,Chart).
```

does the same thing of the previous query, but with particle filtering this time. The results are shown in Figure 12.13b.

Example 12.5.4 Kalman filter cont.
Consider the LPAD and the performed query in Example 12.4.6. We can plot the prior and posterior distribution after we observed `kf(1,[2.5],_T]` *with the query*

```
?- mc_sample_arg(kf(1,_O1,Y),10000,Y,L0),
   mc_lw_sample_arg(kf(1,_O2,T),kf(1,[2.5],_T), 10000,T,L),
   densities(L0,L,40,Chart).
```

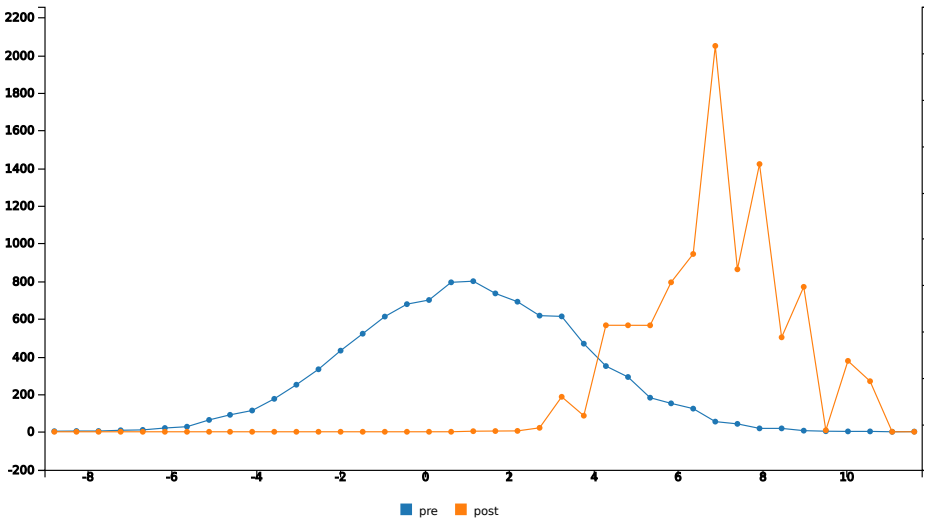
This query that returns the graph of Figure 12.14a, from which it is evident that the posterior distribution is peaked around 2.5.

Given four observations, the value of the state at the same time points can be sampled by running particle filtering:

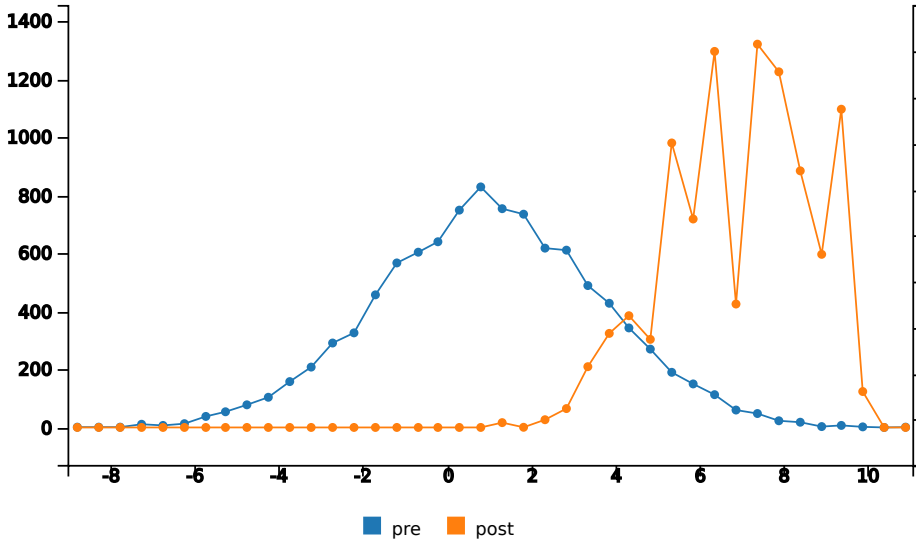
```
?-[O1,O2,O3,O4]=[-0.133, -1.183, -3.212,
   -4.586],
  mc_particle_sample_arg([kf_fin(1,T1),
     kf_fin(2,T2),
  kf_fin(3,T3),kf_fin(4,T4)],[kf_o(1,O1),
     kf_o(2,O2),
  kf_o(3,O3),kf_o(4,O4)],100,[T1,T2,T3,T4],
  [F1,F2,F3,F4]).
```

The list of samples is returned in `[F1,F2,F3,F4]`*, with each element being the sample for a time point.*

Given the states from which the observations were obtained, Figure 12.14b shows a graph with the distributions of the state variable at time 1, 2, 3 and 4 (S1, S2, S3, S4, density on the left Y axis) and with the points for the observations and the states with respect to time (time on the right Y axis).

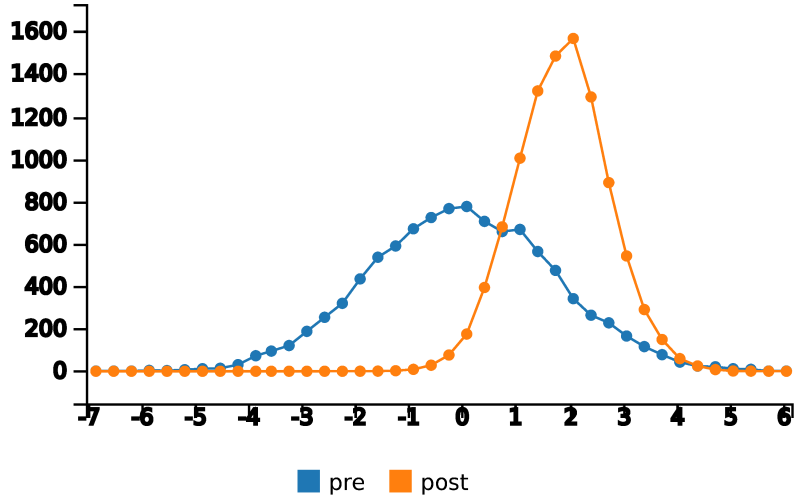


(a) Prior and posterior densities of of the argument `Y` of `value(0,Y)` obtained by likelihood filtering with 10,000 samples (first query in Example 12.5.3).

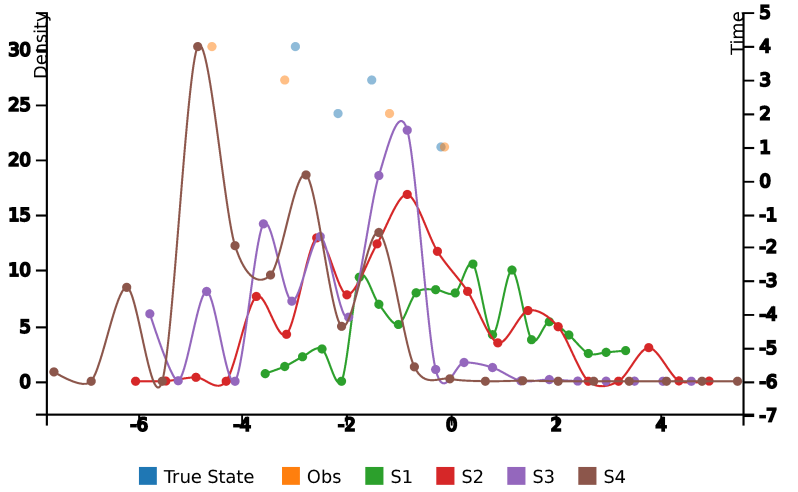


(b) Prior and posterior densities of the argument `Y` of `value(0,Y)` obtained by particle filtering with 10,000 samples (second query in Example 12.5.3).

Figure 12.13: Prior and posterior densities of the argument `Y` of `value(0,Y)` obtained by likelihood weighting and particle filtering in Example 12.5.3



(a) Prior and posterior densities obtained by the first query in Example 12.5.4.



(b) Example of particle filtering obtained by the second query in Example 12.5.4.

Figure 12.14: Representation of the distributions in Example 12.5.4

12.6 Related Work

12.6.1 Work on causality inference

To the best of our knowledge, `cplint` is the first PLP system that allows to perform causal reasoning in an easy, user-friendly and fast way.

P-log [47] is a probabilistic logic programming language that is equipped with a system capable of handling causal reasoning. Differently from LPADs, the semantics of P-log programs is based on Answer Set Programming (ASP) and the possible worlds are the models of the program interpreted as an ASP program. As such, multiple worlds are generated not only because of probabilistic constructs but also because of logical constructs, negation in particular.

The P-log system performs reasoning on such program by computing the whole set of possible worlds using an ASP reasoner. This means enumerating all possible worlds, which can be very expensive. P-log is more suited for programs mixing probabilistic and advanced non-monotonic constructs. If these features are not needed, `cplint` can achieve better results.

Some languages, such as ICL [142] and ProbLog [140], only allow facts as probabilistic clauses. This does not limit the expressiveness, as it is possible to transform an LPAD into an ICL or ProbLog program. For example, the viral marketing program translated into ProbLog is shown in Figure 12.15.

Considering ProbLog as an example, if an action involves a predicate defined only by probabilistic facts, causal inference can be performed by conditional inference. Since probabilistic facts have no parents, in the program above $P(\texttt{has(2)}|do(\texttt{apriori(3)}))$ is equal to $P(\texttt{has(2)}|\texttt{apriori(3)})$ and, at the same time, $P(\texttt{has(2)}|do(\texttt{\textbackslash+apriori(3)}))$ is equal to $P(\texttt{has(2)}|\texttt{\textbackslash+apriori(3)})$. On the other hand, if actions involve predicates defined by rules, as for example in $P(\texttt{has(2)}|do(\texttt{has(3)}))$, the previous simple approach does not apply. In fact, for the action $do(A)$, one should look for all groundings of all probabilistic facts on which A depends and include them in the evidence. This requires a partial evaluation of the program. For the example above one could compute $P(\texttt{has(2)}|do(\texttt{has(3)}))$ by computing $P(\texttt{has(2)}|\texttt{apriori(3)}, \texttt{viral(3,1)}, \texttt{viral(3,2)})$ but in general the partial evaluation may be costly.

Anyway, in case a program can be rewritten by having all predicates for actions defined by facts only, then causal inference can be performed by conditional inference or

```
has(P):- apriori(P).
has(P):- trusts(P, Q), has(Q), viral(P,Q).
apriori(_):0.1.
viral(_,_):0.4.
trusts(2,1).
trusts(3,1).
trusts(3,2).
trusts(4,1).
trusts(4,3).
```

Figure 12.15: ProbLog program for viral marketing.

unconditional inference on simple modifications of the program. This is the approach taken for example in [142], which describes a scenario where there is a robot and a key, the robot can pick up or put down the key and move to different locations[11]. In this example actions are defined only by (certain) facts so their effects can be computed by adding or removing the facts encoding the actions.

The authors of [168] proposed an approach to perform the full *do* calculus on propositional causal models using Answer Set Programming. Moreover, they present an algorithm for inducing models from data. Our approach differs from this because we consider inference for relational causal models, albeit in a restricted case. Therefore our causal random variables can be parameterized by logical variables, as `has(P)` in the viral marketing example: we have a different causal Boolean variable `has(p)` for each person `p` and the rules defining the predicate `has/2` serve as a template for building a complex propositional model of the dependence of `has(p)` from its causes.

12.6.2 Work on Hybrid Probabilistic Logic Programs

A semantics to Hybrid Probabilistic Logic Programs, i.e. logic programs some of the random variables are continuous, was given independently in [160] and [169]. In [161] the semantics of these programs, called Hybrid Probabilistic Logic Programs (HPLP), is defined by means of a stochastic generalization STp of the Tp operator that applies to continuous variables the sampling interpretation of the distribution semantics: STp is applied to interpretations that contain ground atoms (as in standard logic programming) and terms of the form $t = v$ where t is a term indicating a continuous random variable and v is a real number. If the body of a clause is true in an interpretation I, $STp(I)$ will contain a sample from the head. Moreover the authors proposed an evolution of Distributional Clauses called Dynamic Distributional Clauses for dynamic inference with time.

In [169] a probability space for N continuous random variables is defined by considering the Borel σ-algebra over $\mathbb{R}^N$ and a Lebesgue measure on this set as the probability measure. The probability space is lifted to cover the entire program using the least model semantics of constraint logic programs.

12.6.3 Web application for Probabilistic Logic Programming

`cplint` on SWISH is a web application based on SWISH for the `cplint` system. A similar system is ProbLog2 [30], which also has an online version[12]. The main difference between `cplint` on SWISH and ProbLog2 is that the former currently offers also structure learning, approximate conditional inference through sampling and handling of continuous variables. Moreover, `cplint` on SWISH is based on SWISH[13] - a web framework for Logic Programming using features and packages of SWI-Prolog and its Pengines library - and utilizes a Prolog-only software stack in the server, whereas ProbLog2 relies on several different technologies, including Python 3 and the DSHARP

[11] Available also in ProbLog at `https://dtai.cs.kuleuven.be/problog/tutorial/various/14_robot_key.html`.

[12] `https://dtai.cs.kuleuven.be/problog/`

[13] `http://swish.swi-prolog.org`

compiler. In particular, it writes intermediate files to disk in order to call external programs such as DSHARP, while we work in main memory only.

A work strictly related to `cplint` on SWISH is TRILL on SWISH [12]. TRILL on SWISH allows the user to write probabilistic Description Logic (DL) theories in RDF/XML format, and compute the probability of queries with a web browser and is briefly illustrated in Section 13.6.

12.7 Conclusions

In this chapter we presented `cplint`, a PLP system that allows to perform several inference tasks, and its web interface named "`cplint` on SWISH".

In `cplint` causal queries on models with no unknown variables can be answered with exact and approximate inference by exploiting the PITA and MCINTYRE modules respectively. We conducted experiments on the viral marketing problem with random social networks of increasing size. We compared the performance of causal reasoning in `cplint` with regular probabilistic reasoning. The results show that the modification of the inference algorithms do not impact on the execution time and that causal reasoning is in effect cheaper than conditional inference, as expected, thus showing that causal inference is suitable for real life applications.

Like [48], we assume that the causal structure of the model is fully known. Pearl's *do* calculus is more general, as it allows to compute the effect of actions also on models with unknown variables. Exploiting the full power of the *do* calculus in PLP is a very interesting direction for future work.

Hybrid probabilistic logic programs, probabilistic logic programs where some of the random variables are continuous, can also be handled with `cplint` by using the MCINTYRE module, in particular if the evidence contains an atom with continuous values as arguments, you need to use likelihood weighting or particle filtering. Both of them are implemented in the MCINTYRE module of `cplint`, proving that `cplint` is a very useful tool to represent different types of knowledge bases.

Web-based systems are, today, the way to reach out to a wider audience. In order to popularize Probabilistic Logic Programming, we have implemented the web application "`cplint` on SWISH" that allows the user to easily write a Probabilistic Logic Program and compute the probability of queries with a web browser. `cplint` on SWISH already includes a number of examples that cover a wide range of domains and provide interesting applications of Probabilistic Logic Programming. `cplint` on SWISH has been implemented by exploiting the features of the system SWISH for Prolog programming and querying on the Web, and by porting the PITA system for inference on LPADs from its original XSB implementation to SWI-Prolog.

In the next chapter we present algorithms of exact inference for Probabilistic Description Logics that follow DISPONTE.

Chapter 13

Inference in Probabilistic Description Logics

In this chapter we present several algorithms for exact logical inference on Probabilistic Description Logics (PDLs) that follow DISPONTE. Section 13.1 introduces the chapter. Then the following systems are discussed in order: BUNDLE in Section 13.2, TRILL in Section 13.3 and TRILL^P in Section 13.4. Section 13.5 shows how to install TRILL and its derivative systems. Section 13.6 presents a web application for TRILL and TRILL^P. Then a discussion about the complexity of the proposed inference systems is provided in Section 13.7. The experimental results in Section 13.8 show how our systems behave on real world datasets. Related work is discussed in Section 13.9. Finally Section 13.10 draws conclusion and discusses the limitations of our systems and the next steps to undertake.

13.1 Introduction

In Chapter 9 we introduced DISPONTE, a semantics to represent uncertain information with description logics. While in the previous chapter we have illustrated `cplint`, a system that can perform different types of inference on LPADs, in this chapter we show different implemented systems to perform logical probabilistic inference on PDLs that follow DISPONTE. In particular, here we present BUNDLE, TRILL and TRILL^P.

The methods shown in Chapter 11 are still valid for Probabilistic Description Logics that follow DISPONTE. One of the methods for performing logical probabilistic inference is to find the covering set of explanations for a query and make, in some way, all the explanations mutually exclusive. This is the approach used by BUNDLE and TRILL systems. Given a query Q and a KB $\mathcal{K}$, the explanation finding problem has been examined in Subsection 8.2.3 and we defined the problem MIN-A-ENUM which concern the extraction of the set all the possible MinAs ALL-MINAS$(Q, \mathcal{K})$, i.e. the covering set of all the possible minimal explanations.

While BUNDLE is written in Java; TRILL and TRILL^P are written in Prolog.

13.2 BUNDLE

In Subsection 8.2.3 we described how to solve the MIN-A-ENUM problem by means of the hitting set tree (HST) algorithm. This algorithm repeatedly calls a modified tableau algorithm[1] which builds a MinA, i.e. a minimal explanation, from a DL KB from which some axioms are removed depending on the previously found MinAs. BUNDLE [11, 3], for "Binary decision diagrams for Uncertain reasoNing on Description Logic thEories", computes the probability of query from a KB that follows DISPONTE by first finding a covering set of explanations and then making them pairwise incompatible by using BDDs (see Section 11.5). To solve the MIN-A-ENUM BUNDLE uses the HST algorithm and, every time a new MinA is needed, BUNDLE exploits Pellet reasoner and its tableau algorithm [82]. Finally, it computes the probability from the BDD by using function PROB of Algorithm 11.2.

Algorithm 13.1 shows BUNDLE's main procedure. It first builds a data structure $PMap$ that associates each probabilistic DL axiom E_i with its probability p_i (line 8). If E_i is associated with more than a probability, BUNDLE aggregates all the values following the semantics (see Example 9.2.4), the resulting probability is inserted in $PMap$. Then it computes the MinAs for the query Q by calling the EXPHST function (line 9) which in turn executes the HITTINGSETTREE function[2], shown in Algorithm 8.4. EXPHST can also take as input several parameters such as the maximum number of explanations to be generated $maxEx$ and the time limit $maxTime$ for the inference process. If one of the limits is reached during the execution of the HST algorithm, EXPHST stops and returns the set of explanations found so far.

Two data structures are initialized: $VarAx$ is an array that contains the association between Boolean random variables (whose index is the array index) and pairs (axiom, probability), and BDD stores a BDD. BDD is initialized to the `zero` Boolean function (lines 10-11).

Then BUNDLE builds a BDD representing the set of explanations by means of two nested loops (lines 12-29). In the outer loop it iterates over explanations, whereas in the inner loop it iterates over the axioms that compose the current explanation. JavaBDD[3] [132] is exploited to manipulate BDDs. As mentioned in Section 10.3, it is a Java BDD library and an interface to a number of underlying BDD manipulation packages. The underlying package to use can be dynamically chosen by means of a specific argument, by default BuDDy is used.

In the inner loop, BUNDLE generates the BDD for a single explanation, indicated as $BDDE$, which is initialized to the `one` Boolean function (lines 14-27). The axioms of each MinA are considered one by one. If the axiom is certain, then the `one` Boolean function is stored in $BDDA$ (line 16). Otherwise, the axiom Ax is searched for in $PMap$ and the associated probability value p is extracted. The axiom is also searched for in $VarAx$ to check whether a random variable has already been assigned to it (lines 18-19). If not, a cell is added to $VarAx$ to store the pair (line 21). At this point we know the position i of the pair (Ax, p) in the array $VarAx$, that is the index of its Boolean

[1] The HST algorithm is not restricted to tableau algorithms, it can be used with any algorithm that retrieves a MinA given a query.

[2] As mentioned before the MinAs correspond to the conflict sets found by the HST Algorithm.

[3] Available at `http://javabdd.sourceforge.net/`

Algorithm 13.1 Function BUNDLE: computation of the probability of a query Q given the (probabilistic) KB $\mathcal{K}$.

```
 1: function BUNDLE(Q, K, maxEx, maxTime)
 2:     Input: Q (the query (a concept) to be tested for satisfiability)
 3:     Input: K (the knowledge base)
 4:     Input: maxEx (the maximum number of explanations to find)
 5:     Input: maxTime (the time limit for the inference)
 6:     Output: the set of explanations (MinAs) found for the unsatisfiability of Q w.r.t. K
 7:     Output: the probability of the query Q w.r.t. K
 8:     Build Map PMap with sets of pair (axiom, probability)
 9:     MinAs ←EXPHST(Q, K, maxEx, maxTime)
10:     Initialize VarAx to empty          ▷ VarAx is an array of pairs (axiom, probability)
11:     BDD ←BDDZERO
12:     for all MinA ∈ MinAs do
13:         BDDE ←BDDONE
14:         for all Ax ∈ MinA do
15:             if Ax in K is a certain axiom then
16:                 BDDA ←BDDONE
17:             else
18:                 p ← PMap(Ax)
19:                 Scan VarAx looking for Ax
20:                 if !found then
21:                     Add to VarAx a new cell containing (Ax, p)
22:                 end if
23:                 Let i be the position of (Ax, p) in VarAx
24:                 BDDA ← BDDGETITHVAR(i)
25:             end if
26:             BDDE ←BDDAND(BDDE,BDDA)
27:         end for
28:         BDD ←BDDOR(BDD,BDDE)
29:     end for
30:     queryProb ←PROB(BDD, ∅)          ▷ VarAx is used to compute P(X) in PROB
31:     return (MinAs, queryProb)
32: end function
```

random variable X_i. We obtain a BDD representing $X_i = 1$ with BDDGETITHVAR in $BDDA$ (line 24). $BDDA$ is finally conjoined with the current $BDDE$ to get the BDD representing a single explanation (line 26).

In the outermost loop, BUNDLE combines BDDs for different explanations through disjunction between BDD and the current explanation $BDDE$ (line 28).

After the two cycles, the BDD BDD is fully built. we can finally invoke function PROB of Algorithm 11.2 to compute the probability of the query from BDD.

In [3] the authors proved BUNDLE correctness.

Theorem 13.1 BUNDLE correctness ([3])
Given a DISPONTE knowledge base $\mathcal{K}$, a query Q and one or both limits $maxEx$ and $maxTime$ for the number of explanations to find and for the inference time respectively, the probability returned by BUNDLE, BUNDLE$(Q, \mathcal{K}, maxEx, maxTime)$ *could be:*

- *A lower bound on $P(Q)$ if a maximum number of explanations to compute and/or a time limit are set and at least one of the limits is reached, i.e.,*

$$\text{BUNDLE}(Q, \mathcal{K}, maxEx, maxTime) \leq P(Q)$$

This means that we have computed a subset of the covering set of minimal explanations for the query Q. This is a direct consequence of Theorem 11.3.

- *Equal to $P(Q)$, i.e.,*

$$\text{BUNDLE}(Q, \mathcal{K}, maxEx, maxTime) = P(Q)$$

This means that we have computed all the explanations of the query Q and therefore we have obtained the exact probability of Q. This is a direct result of Theorem 11.2.

13.2.1 How to use BUNDLE

The latest stable version of BUNDLE is 2.3.1 and it can be downloaded at `https://bitbucket.org/machinelearningunife/bundle/downloads/`. It can be used as standalone desktop application. The manual to use BUNDLE can be found at `https://bitbucket.org/machinelearningunife/bundle/wiki/Home`.

If your application needs to perform probabilistic logical inference, BUNDLE can also be used as a library. If you are using Maven, all you have to do is to add the following lines in your `POM.xml`.

```
<dependency>
   <groupId>it.unife.endif.ml</groupId>
   <artifactId>bundle</artifactId>
   <version>2.3.1</version>
</dependency>
```

Then you can obtain the probability of query in just few lines:

```
Bundle reasoner = new Bundle();
reasoner.setRootOntology(rootOntology);
reasoner.computeQuery(query);
```

where `rootOntology` and `query` are objects of the classes `OWLOntology` and `OWLAxiom` of OWLAPI library respectively.

BUNDLE has been integrated in DL-Learner 1.3 [170], together with EDGE (see Chapter 16) and LEAP (see Chapter 18).

13.3 TRILL

In Subsection 8.2.2 we discussed Pellet's tableau algorithm (Algorithm 8.1), and its expansions rules (Figure 8.1). Some of these expansion rules are non-deterministic and in order to find all the explanations all the non-deterministic choices must be explored. The HST algorithm, discussed in Section 8.2.3, is a way for procedural languages, such as Java, to manage with the non-determinism of the tableau.

Prolog is a declarative language that natively supports non-determinism by means of backtracking facilities. This led to the development of TRILL.

TRILL stands for "Tableau Reasoner for descrIption Logics in Prolog", it implements the tableau algorithm in Prolog, so the management of the non-determinism of the rules

is delegated to the language, and solves MIN-A-ENUM. It can answer concept and role membership queries, subsumption queries and can test the unsatisfiability of a concept of the KB or the inconsistency of the entire KB.

The code of TRILL is available at `https://github.com/rzese/trill`.

We use the Thea2 library [171] for converting OWL DL KBs into Prolog. Thea2 performs a direct translation of OWL axioms into Prolog facts. For example, a simple subclass axiom between two named classes $\mathsf{Cat} \sqsubseteq \mathsf{Pet}$ is written using the `subClassOf/2` predicate as `subClassOf('Cat','Pet')`. For more complex axioms, Thea2 exploits the list construct of Prolog, so the axiom

$$\mathsf{NatureLover} \equiv \mathsf{PetOwner} \sqcup \mathsf{GardenOwner}$$

becomes

```
equivalentClasses(['NatureLover', unionOf(['PetOwner','GardenOwner'])]).
```

In order to represent the tableau, TRILL uses a pair $Tableau = (A, T)$, where

- A represents the ABox, it is a list containing information about individuals and class assertions with the corresponding value of the tracing function, i.e. the corresponding set of explanations. An example of ABox A is shown in Figure 13.1, which states that kevin is a nominal (`nominal('kevin')`) and that it belongs to concept Man and to concept Person.

```
[ (classAssertion('Person','kevin'),
    [subClassOf('Man','Person'),
   classAssertion('Man','kevin')]),
    (classAssertion('Man','kevin'),[
       classAssertion('Man','kevin
       ')]),
    nominal('kevin') ]}
```

Figure 13.1: Example of ABox in TRILL

- T is a triple (G, RBN, RBR) in which G is a directed graph that contains the structure of the tableau, RBN is a red-black tree (a key-value dictionary), where a key is a couple of individuals and its value is the set of the labels of the edge between the two individuals, and RBR is a red-black tree, where a key is a role and its value is the set of couples of individuals that are linked by the role. This representation allows to quickly find the information needed during the execution of the tableau algorithm.

For managing the *blocking* system we use a predicate for each blocking state: `nominal/2`, `blockable/2`, `blocked/2`, `indirectly_blocked/2` and `safe/3`. Each predicate takes as arguments the individual `Ind` and the tableau `(ABox, Tab)`. `safe/3`, shown in Figure 13.2, takes as input also the role `R` ; `rb_lookup/3` looks for a pair of individuals connected by the role `R` and `neighbors/3` returns the list of neighbors of `Ind` in `N`.

```
safe(Ind,R,(ABox,Tab)):-
    (T,RBN,RBR) = Tab,
    rb_lookup(R,V,RBR),
    member((X,Ind),V),
    blockable(X,(ABox,(T,RBN,RBR)))
       ,!.

safe(Ind,R,(ABox,Tab)):-
    (T,RBN,RBR) = Tab
    rb_lookup(R,V,RBR),
    member((X,Ind),V),
    nominal(X,(ABox,(T,RBN,RBR))),!,
    \+ blocked(Ind,(ABox,(T,RBN,RBR)
       )).
```

Figure 13.2: Code of the predicate `safe/3`. An `R`-neighbour `Ind` of `X` is safe if (i) `X` is blockable - corresponding with the first definition, where the predicate `blockable/2` is called - or if (ii) `X` is a nominal node and `Ind` is not blocked - checked by `nominal/2` and `blocked/2` -. The predicates `rb_lookup/3` and `member/2` are used to find an `R`-predecessor `X` to check if `Ind` is safe following the definition.

During initialization of the tableau, for each individual `Ind` in the ABox, we add the atom `nominal(Ind)` to `A`, then every time we have to check the blocking status of an individual we call the corresponding predicate that returns the status by checking the tableau.

Deterministic and non-deterministic tableau expansion rules are implemented following a different interface; this will facilitate the insertion of new rules in the future.

Deterministic rules are implemented by a predicate `rule_name(Tab,Tab1)` that, given the current tableau `Tab`, returns the tableau `Tab1` obtained by the application of the rule to `Tab`.

Figure 13.3 shows part of the code of the deterministic rule $\rightarrow$ *unfold*. The predicate `unfold_rule/2` searches in `(ABox, Tab)` for an individual to which the rule can be applied and calls the predicate `find_sub_sup_class/3` in order to find the class to be added to the label of the individual.

All non-deterministic rules are implemented following the interface `rule_name(Tab, TabList)`, thus they take as input the current tableau `Tab` and return the list of tableaux `TabList` created by the application of the rule to `Tab`.

Figure 13.4 shows the code of the non-deterministic rule $\rightarrow \sqcup$. The predicate `or_rule/2` searches in the tableau `Tab0`, which corresponds to the pair `(ABox0,Tabs0)`, for an individual to which the rule can be applied and unifies `L` with the list of new tableaux created by `scan_or_list/6`. `find/2` implements the search for a class assertion. Since the data structure that stores class assertions is currently a list, `find/2` simply calls `member/2`. `absent/3` checks if the class assertion axiom with the associated explanation is already present in `ABox`, and in this case it checks the applicability of the expansion rule.

```
unfold_rule((A,T),([(classAssertion(D,Ind),[(Ax,Ind)|Expl])|A],T)):-
  find((classAssertion(C,Ind),Expl),A),
  atomic(C),
  find_sub_sup_class(C,D,Ax),
  absent(classAssertion(D,Ind),[(Ax,Ind)|Expl],(A,T)).

find_sub_sup_class(C,D,subClassOf(C,D)):-
  subClassOf(C,D).

find_sub_sup_class(C,D,equivalentClasses(L)):-
  equivalentClasses(L),
  member(C,L),
  member(D,L),
  dif(C,D).
```

Figure 13.3: Code of the $\rightarrow$ *unfold* rule. It takes an atomic class `C` from the input tableau and looks for a class `D` which is a superclass or an equivalent class of `C` and it is not already in the tableau, builds the explanation for the new class assertion found and adds it to the resulting tableau.

```
or_rule((ABox0,Tabs0),L):-
  find((classAssertion(unionOf(LC),Ind),Expl),ABox0),
  \+indirectly_blocked(Ind,(ABox0,Tabs0)),
  findall((ABox1,Tabs0),scan_or_list(LC,Ind,
        Expl,ABox0,Tabs0,ABox1),L),
    dif(L,[]),!.

scan_or_list([],_Ind,_Expl,ABox,_Tabs,ABox).

scan_or_list([C|_T],Ind,Expl,ABox,Tabs,
  [(classAssertion(C,Ind),Expl)|ABox]):-
  absent(classAssertion(C,Ind),Expl,(ABox,Tabs)).

scan_or_list([_C|T],Ind,Expl,ABox0,Tabs,ABox):-
   scan_or_list(T,Ind,Expl,ABox0,Tabs,ABox).
```

Figure 13.4: Code of the $\rightarrow \sqcup$ rule. It unifies the list `L` with all the tableau resulting by the application of the rule.

Expansion rules are applied in order by `apply_all_rules/2`, first the deterministic ones and then the non-deterministic ones, as shown in Figure 13.5. The predicate `apply_det_rules/3` takes as input the list of deterministic rules and the current tableau and returns a tableau obtained by the application of one of the rules. It is called as `apply_det_rules(RuleList,Tab,Tab1)`. After the application of a deterministic rule, a cut avoids backtracking to other possible choices for the deterministic rules.

Then, non-deterministic rules are tried sequentially with `apply_nondet_rules/3`, shown in Figure 13.5, that is called as `apply_nondet_rules(RuleList, Tab, Tab1)`. It takes as input the list of non-deterministic rules and the current tableau and returns a tableau obtained with the application of one of the rules. If a non-deterministic rule is applicable, the list of tableaux obtained by its application is returned by the predicate corresponding to the applied rule, a cut is performed to avoid backtracking to other rule choices and a tableau from the list is non-deterministically chosen with the `member/2` predicate. If no rule is applicable, the input tableau is returned and the rule application stops, otherwise a new round of rule application is performed.

Finally, the `findall/3` predicate is used on the set of the built tableaux for finding all the clashes contained in them in order to collect all the possible explanations.

In each rule application round, the applicability of a rule is checked by looking whether its result is not already present in the tableau. This avoids both infinite loops in the rule application and considering alternative rules when a rule is applicable. In fact, if a rule is applicable in a tableau, it will also be so in any tableau obtained by the expansion of the original one. In this case, the choice of which expansion rule to apply introduces "don't care" non-determinism. Differently, "don't know" non-determinism is introduced by non-deterministic rules, since a single tableau is expanded into a set of tableaux. We use Prolog search only to handle "don't know" non-determinism.

In Figure 8.1, the symbol $(*)$ denotes the rules used by TRILL. When a concept is already present in a node label, TRILL checks whether to update the tracing function by verifying that the corresponding set of axioms is not a subset of τ.

TRILL is limited to to $\mathcal{SHIQ}$ KBs. In particular we can notice that the NN-rule [93] is not implemented. This rule is necessary if the DL KB involves inverse roles, number restrictions and nominals.

When the set of covering explanations is found, TRILL computes the probability of a query by means of the `compute_prob/2` predicate, shown in Figure 13.6. `build_bdd/3` takes each explanation in `Expls` and, for each axiom in the explanations, looks for its probability using `get_prob_ax/3` (certain axioms have probability 1). `Expls` is a list similar to the ABox in Figure 13.1.

The probability of the query is finally computed from the newly built BDD by using `ret_prob/3`. `one/2` and `zero/2` return BDDs representing the Boolean constants 1 and 0; `and/4` and `or/4` perform the AND and OR Boolean operations between BDDs. `get_var_n/5` returns the random variable `V` associated with axiom `AxN` and the list of probabilities `[Prob,ProbN]`, where $ProbN = 1 - Prob$. `equality/4` returns the BDD `BDDeq` associated with the expression $V = val$ where V is a random variable and val is 0 or 1. The `ret_prob/3`, `one/2`, `zero/2`, `and/4`, `or/4` and `equality/4` predicates are the same predicates defined in Subsection 12.2.1.

```
apply_all_rules(Tab0,Tab):-
  apply_det_rules(
    [o_rule,and_rule,unfold_rule,
       add_exists_rule,forall_rule,
       forall_plus_rule,exists_rule,
       min_rule],
    Tab0, Tab1),
  (Tab0=Tab1 *->
     Tab=Tab1;
     apply_all_rules(Tab1,Tab)
  ).

apply_det_rules([],Tab0,Tab):-
  apply_nondet_rules([or_rule,max_rule],
     Tab0,Tab).

apply_det_rules([H|_],Tab0,Tab):-
  call(H,Tab0,Tab),!.

apply_det_rules([_|T],Tab0,Tab):-
  apply_det_rules(T,Tab0,Tab).

apply_nondet_rules([],Tab,Tab).

apply_nondet_rules([H|_],Tab0,Tab):-
  call(H,Tab0,L),!,
  member(Tab,L),
  dif(Tab0,Tab).

apply_nondet_rules([_|T],Tab0,Tab):-
  apply_nondet_rules(T,Tab0,Tab).
```

Figure 13.5: Application of the expansion rules by means of the predicates `apply_all_rules/2`, `apply_det_rules/3` and `apply_nondet_rules/3`. The list `[o_rule,and_rule,...]` contains the available rules in TRILL.

```
compute_prob(Expl,Prob):-
  get_number_of_probabilistic_axioms(NV),
  init_test(NV,Env),
  build_bdd(Env,Expls,BDD),
  ret_prob(Env,BDD,Prob),
  end_test(Env), !.

build_bdd(Env,[X],BDD):- !,
  bdd_and(Env,X,BDD).

build_bdd(Env, [H|T],BDD):-
  build_bdd(Env,T,BDDT),
  bdd_and(Env,H,BDDH),
  or(Env,BDDH,BDDT,BDD).

build_bdd(Env,[],BDD):- !,
  zero(Env,BDD).

bdd_and(Env,[X],BDDeq):-
  get_prob_ax(X,AxN,Prob),!,
  ProbN is 1-Prob,
  get_var_n(Env,AxN,[],[Prob,ProbN],V),
  equality(Env,V,0,BDDeq),!.

bdd_and(Env,[_X],BDDX):- !,
  one(Env,BDDX).

bdd_and(Env,[H|T],BDDAnd):-
  get_prob_ax(H,AxN,Prob),!,
  ProbN is 1-Prob,
  get_var_n(Env,AxN,[],[Prob,ProbN],V),
  equality(Env,V,0,BDDeq),
  bdd_and(Env,T,BDDT),
  and(Env,BDDeq,BDDT,BDDAnd).

bdd_and(Env,[_H|T],BDDAnd):- !,
  one(Env,BDDH),
  bdd_and(Env,T,BDDT),
  and(Env,BDDH,BDDT,BDDAnd).
```

Figure 13.6: Code of the predicates `compute_prob/2` and `build_bdd/3`.

13.4 TRILL^P

TRILL^P stands for "Tableau Reasoner for descrIption Logics in Prolog powered by Pinpointing formula". It is based on the reasoner TRILL, but instead of resolving MIN-A-ENUM like TRILL, it builds a **pinpointing formula** representing the set of all MinAs for a given query, following the approach defined in Subsection 8.2.4.

Since TRILL^P is based on TRILL, the representation of the tableau and the management of the blocking system are the same of TRILL. However, there are some differences due to the different representation of the explanations. Both TRILL and TRILL^P represent the tableau as a couple $Tableau = (A, T)$, but in TRILL^P the label of each class assertion in the list A contains the pinpointing formula instead of the explanations.

The pinpointing formula is encoded as a combination of predicates `*/1` and `+/1`. For instance, the Boolean formula $((E_2 \wedge E_4) \vee (E_3 \wedge E_5)) \wedge E_6 \wedge E_1$ from Example 8.2.3 is modeled as `*([+([*([E2,E4]),*([E3,E5])]),E6,E1])`.

As mentioned in Subsection 8.2.4, the algorithm used for computing the pinpointing formula is limited to $\mathcal{SHI}$ KBs, therefore TRILL^P uses a subset of TRILL's expansion rules. In particular, it uses the rules defined in Figure 8.4.

The $\psi-$*insertability* test is done by means of a satisfiability solver. In particular, TRILL^P conjoins the negation of the pinpointing formula $lab(a)$ contained in the label of the individual in the tableau with the new Boolean formula ψ to add to the label and tests the satisfiability of such formula. In other words to check whether $\psi \models lab(a)$, TRILL^P checks whether $\psi \wedge \neg lab(a)$ is satisfiable. If the formula is satisfiable, i.e. $\psi \not\models lab(a)$ then the assertion is ψ-insertable. This step is performed by the `test/2` predicate shown in Figure 13.7. The predicate `test/2` first calls `build_f/3` which takes two Boolean formulas `L1` (ψ) and `L2` ($lab(a)$) and creates the conjunction that will be tested by means of the satisfiability solver. Predicates `cnf/2` and `sat/1` are Prolog libraries providing an interface to a SAT solver. Predicate `cnf/2` converts a propositional formula `F`, into a Conjunctive Normal Form (CNF) `Cnf`. Finally, `sat/1` takes as input such a CNF formula and succeeds if it is satisfiable. If the test returns true, TRILL^P combines the two Boolean formulas with the `OR` Boolean operator. The YAP version of `sat/1` exploits a library [172] that interfaces with the MiniSat SAT solver [173], a small (about 1200 lines of C code) and efficient sat-solver; whereas the SWI-Prolog version exploits the solver contained in the `clpb`[4] library.

[4] `http://www.swi-prolog.org/pldoc/man?section=clpb`

```
test(L1,L2):-
  build_f(L1,L2,F),
  cnf(F,Cnf),
  sat(Cnf).

build_f([L1],[L2],(F1*(-F2))):-
  build_f1(L1,F1,[],Var1),
  build_f1(L2,F2,Var1,_Var).
```

Figure 13.7: Definition of the predicates `test/2` and `build_f/3`. In particular, `build_f1/4` takes as input a Boolean formula (the first argument) and a list containing the correspondence between Boolean variables and axioms (the third argument) and returns the Boolean formula in a format suitable for the predicate `cnf/2` (the second argument) and the updated correspondence list.

13.5 How to use TRILL and TRILL^P

TRILL and TRILL^P are available both for Yap Prolog[5] [151] and in SWI-Prolog[6] [152]. Using SWI-Prolog, with the goal `pack_install(trill)` the user will install all these systems. After the installation, they can be loaded with the command `use_module(library(trill)).` The code is available at `https://github.com/rzese/trill`.

13.6 TRILL on SWISH

In order to popularize DISPONTE, we developed a Web application called **TRILL on SWISH** and available at `http://trill.lamping.unife.it`. TRILL on SWISH is strictly related to `cplint` on SWISH. As for `cplint` on SWISH, we exploited SWISH [165], a recently proposed Web framework for logic programming that is based on various features and packages of SWI-Prolog (see Subsection 12.5.1).

We modified it in order to manage OWL KBs. TRILL-on-SWISH allows users to write a KB in the RDF/XML format directly in the web page or load it from a URL, and specify queries that are answered by TRILL running on the server. Once the computation ends, the results are sent to the client browser and visualized in the Web page.

Figure 13.8 shows the interface of TRILL on SWISH. We can notice that is very similar to the interface of `cplint` on SWISH.

[5] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[6] `http://www.swi-prolog.org/`

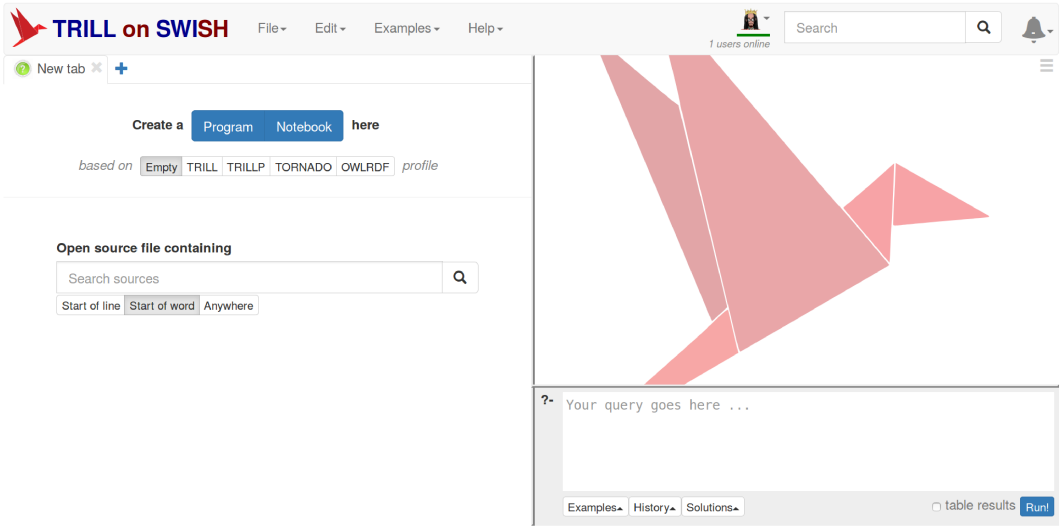


Figure 13.8: TRILL on SWISH interface.

13.7 Inference Complexity

The computational complexity of computing the probability of an axiom can be studied by considering the two problems that must be solved.

- The first problem is that of *explanation finding* or *axiom pinpointing*. Computational complexity of the MIN-A-ENUM problem has been studied in a number of works [174, 175, 176]. Baader et al. [177] showed that there can be exponentially many MinAs even for very simple DLs that allows only concept intersection[7] ($\sqcap$). Thus, for more expressive DLs such as $\mathcal{SHIQ}(\mathbf{D})$ (and $\mathcal{SROIQ}(\mathbf{D})$), the number of explanations for $\mathcal{SHIQ}(\mathbf{D})$ (and $\mathcal{SROIQ}(\mathbf{D})$) may be even larger. Given this fact, we do not consider complexity with respect to the input only. We say that an algorithm runs in output polynomial time [178] if it computes all the output in time polynomial in the overall size of the input and the output. Corollary 15 in [176] shows that MIN-A-ENUM cannot be solved in *output polynomial time* for *DL-Lite*$_{bool}$ TBoxes unless $P = NP$, i.e., MIN-A-ENUM is not polynomial in the size of the TBox and the number of MinAs. *DL-Lite*$_{bool}$ is less expressive than $\mathcal{SHIQ}(\mathbf{D})$ (and $\mathcal{SROIQ}(\mathbf{D})$), thus these results also hold for $\mathcal{SHIQ}(\mathbf{D})$ (and $\mathcal{SROIQ}(\mathbf{D})$). Moreover, Corollary 3 in [100] shows that a pinpointing formula for the unsatisfiability of a concept w.r.t. an $\mathcal{ALC}$ KB can be computed in time exponential in the size of the input.

- The second problem is computing the probability of a SUM-OF-PRODUCTS, that is

  Definition 13.1 SUM-OF-PRODUCTS
  Given a Boolean expression S in disjunctive normal form (DNF) or a SUM-OF-PRODUCTS in the variables $\{V_1, \ldots, V_n\}$ and $P(V_i)$, the probability that V_i is true with $i = 1, \ldots, n$, compute the probability $P(S)$ of S, assuming all variables are independent. □

[7] The authors of [177] called this DL $\mathcal{HL}$.

We have already seen that the input of the SUM-OF-PRODUCTS problem is of at least exponential size in the worst case, moreover SUM-OF-PRODUCTS was shown to be #P-hard [179], hence computing the probability of an axiom from a $\mathcal{SHIQ}(\mathbf{D})$ (and $\mathcal{SROIQ}(\mathbf{D})$) knowledge base is intractable. However, the algorithms that have been proposed for solving the two problems proved to be able to work on input of real world size. For example, all MinAs have been found for various entailments over many real world ontologies within a few seconds [90, 95]. Concerning the SUM-OF-PRODUCTS problem, algorithms based on BDDs were able to solve problems with hundreds of thousands of variables (see e.g. the works on inference on probabilistic logic programs [45, 180, 27, 181, 140, 182, 144, 49, 183]). Also methods for weighted model counting [148, 184] can be used to solve the SUM-OF-PRODUCTS problem.

Moreover, Section 13.8 shows that in practice our algorithms can compute the probability of entailments on KBs of real-world size.

13.8 Experiments

In [185, 3] BUNDLE was compared with PRONTO by running queries w.r.t. increasingly complex ontologies. The experiments showed that BUNDLE can manage larger KBs than PRONTO due to the low amount of memory needed and it is faster. Starting from these results we compared our system taking BUNDLE as basis for comparison. The experiment were performed on a Linux machine with an Intel Core 2 Quad CPU Q6600 @ 2.40GHz.

13.8.1 Comparing the Systems

We conducted two different experiments to compare the inference times spent to answer a probabilistic query by our systems BUNDLE, TRILL and TRILL^P[8].

For the first test we used four real world KBs of various complexity as in [10]:

- BRCA[9], which models the risk factors of breast cancer;
- an extract of the DBPedia[10] ontology obtained from Wikipedia;
- Biopax level 3[11], which models metabolic pathways;
- Vicodi[12], which contains information on European history.

BRCA ("Breast Cancer Risk Assessment") [186] states the risk factors of breast cancer depending on several factors such as age, drugs taken, ethnicity, etc. For example, as shown in Figure 13.9, it states that known factors for the disease are the lack of exercise or having the first child at a late age. The original version presented in [186]

[8] We used the SWI-Prolog version of TRILL and TRILL^P.

[9] `http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl`

[10] `http://dbpedia.org/`

[11] `http://www.biopax.org/`

[12] `http://www.vicodi.org/`

```
...
<owl:Class rdf:about="&cancer_ra;
   LackOfExercise">
  <rdfs:subClassOf rdf:resource="&cancer_ra
     ;KnownFactor"/>
</owl:Class>

<owl:Class rdf:about="&cancer_ra;
   LateFirstChild">
  <rdfs:subClassOf rdf:resource="&cancer_ra
     ;KnownFactor"/>
</owl:Class>
...
```

Figure 13.9: Axioms from BRCA. They state that the lack of exercise may increase the risk of having breast cancer as well as having the first child at old age.

```
...
<owl:Class rdf:about="&dbpedia;
   ComicsCreator">
  <rdfs:subClassOf rdf:resource="&dbpedia;
     Person"/>
</owl:Class>
...
<owl:Class rdf:about="&dbpedia;Hospital">
  <rdfs:subClassOf rdf:&dbpedia;Place"/>
</owl:Class>
...
```

Figure 13.10: Axioms from DBPedia. They state that a comic creator is a person and that hospitals are places.

reduced risk assessment to probabilistic entailment in P-$\mathcal{SHIQ}(\mathbf{D})$. For this test we took only the non-probabilistic part and we made probabilistic 50 axioms randomly chosen. BRCA has $\mathcal{ALCHF}(\mathbf{D})$ expressiveness and contains 491 axioms, 154 different classes and 15 different properties (12 object properties and 3 data properties).

DBPedia contains structured information about Wikipedia. It is built on the data contained in the sideboxes shown in wiki pages. Some examples of axioms are shown in Figure 13.10. DBPedia's expressiveness is $\mathcal{EL}$. The portion we consider contains 267 axioms and 118 classes.

Biopax [187] was defined in order to allow integration on analysis of biological pathways data. The BioPax project defines 3 different levels, in this test we used level 3 that represents molecular and genetic interactions together with pathways including molecular states. Figure 13.11 shows the axioms asserting that physical entities are composed of complexes which are in turn physical entities different from DNAs, RNAs

Table 13.1: Average time of BUNDLE, TRILL and TRILL^P for different datasets.

	BioPax	BRCA	DBPedia	Vicodi
TRILL	0.113	**0.572**	22.972	**0.034**
TRILL^P	**0.078**	1.083	4.764	**0.026**
BUNDLE	1.85	6.96	**3.79**	1.12

Table 13.2: Average time (in seconds) for computing the probability of queries with the reasoners BUNDLE, TRILL and TRILL^P for synthetic datasets. The cells containing "–" mean that the execution was timed out (600 s). In particular, with $n = 10$, TRILL^P took more than 24 hours.

	2	4	6	8	10
TRILL	**0.004**	**0.102**	7.027	556.927	–
TRILL^P	0.018	0.680	93.546	–	–
BUNDLE	0.23	0.733	**6.032**	–	–

or small molecules. The version of Biopax used has $\mathcal{SHIN}(\mathbf{D})$ expressiveness and has 925 axioms, 69 classes, 55 object properties and 41 data properties.

Finally, Vicodi [188] is an extract of the Vicodi knowledge base that contains information on European history. It models historical events and important personalities such as popes or princes, as shown in Figure 13.12. Vicodi's expressiveness is $\mathcal{ALH}(\mathbf{D})$ and contains 209 axioms, 168 classes, 6 object properties and 2 data properties

We used a version of the DBPedia and Biopax KBs without the ABox and a version of BRCA and Vicodi with an ABox containing 1 individual and 19 individuals respectively. For each KB we added a probability annotation to each axiom. We randomly created 50 subclass-of queries for DBPedia and Biopax and 50 instance-of queries for the other KBs, ensuring each query has at least one explanation.

Table 13.1 shows the average time in seconds to answer the queries for BUNDLE, TRILL and TRILL^P. In particular, BRCA and DBPedia contain many subclass axioms between complex concepts, resulting in many explanations. On BRCA, TRILL^P performs worse than TRILL because the SAT solver is called many times with complex formulas. As can be seen, the SAT solver has a large impact on performances.

In the second experiment we used the KB of Example 8.2.4 where all the axioms were made probabilistic, annotating them with a random value of probability. We increased n from 2 to 10 in steps of 2 and we collected the running time, averaged over 50 executions. Table 13.2 shows, for each n, the average time in seconds that the systems took for computing the probability of the query Q. For the computation we set a timeout of 10 minutes for each query execution, so the cells with "–" indicate that the time out occurred.

```
...
<owl:ObjectProperty rdf:about="&biopax;
   component">
  <rdf:type rdf:resource="&owl;
     InverseFunctionalProperty"/>
  <rdfs:domain rdf:resource="&biopax;
     Complex"/>
  <rdfs:range rdf:resource="&biopax;
     PhysicalEntity"/>
</owl:ObjectProperty>
...
<owl:Class rdf:about="&biopax;Complex">
  <rdfs:subClassOf rdf:resource="&biopax;
     PhysicalEntity"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&biopax
         ;memberPhysicalEntity"/>
      <owl:allValuesFrom rdf:resource="&
         biopax;Complex"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="&biopax;
     Dna"/>
  <owl:disjointWith rdf:resource="&biopax;
     DnaRegion"/>
  <owl:disjointWith rdf:resource="&biopax;
     Protein"/>
  <owl:disjointWith rdf:resource="&biopax;
     Rna"/>
  <owl:disjointWith rdf:resource="&biopax;
     RnaRegion"/>
  <owl:disjointWith rdf:resource="&biopax;
     SmallMolecule"/>
  ...
</owl:Class>
...
```

Figure 13.11: Axioms from Biopax. A complex is a physical entity different from DNAs, RNAs or their regions, proteins and small molecules. It can be part of another physical entity.

```
...
<owl:Class rdf:about="&vicodi;Pope">
  <rdfs:subClassOf rdf:resource="&vicodi;
     Religious-Leader"/>
</owl:Class>
...
<owl:Class rdf:about="&vicodi;Prince">
  <rdfs:subClassOf rdf:resource="&vicodi;
     Head-of-State"/>
</owl:Class>
...
<owl:Class rdf:about="&vicodi;War">
  <rdfs:subClassOf rdf:resource="&vicodi;
     Event"/>
</owl:Class>
...
```

Figure 13.12: Axioms from Vicodi. A Pope is a religious leader, while a prince is a head of a state. Finally a war is an historical event.

13.9 Related Work

Usually, DL reasoners implement a tableau algorithm using a procedural language. Since some tableau expansion rules are non-deterministic, the developers have to implement a search strategy from scratch. Moreover, in order to solve MIN-A-ENUM, all different ways of entailing an axiom must be found.

Reasoners written in Prolog can exploit its backtracking facilities for performing the search. This has been observed in various works. In [189] was proposed a tableau reasoner in Prolog for FOL based on free-variable semantic tableaux. However, the reasoner is not tailored to DLs. Meissner [190] presented the implementation of a Prolog reasoner for the DL $\mathcal{ALCN}$. This work was the basis of [191], that considered $\mathcal{ALC}$ and improved [190] by implementing heuristic search techniques to reduce the running time. In [192] was added to [191] the possibility of returning explanations for queries but it still handled only $\mathcal{ALC}$.

In [88] the authors presented the KAON2 algorithm that exploits basic superposition, a refutational theorem proving method for FOL with equality, and a new inference rule, called decomposition, to reduce a $\mathcal{SHIQ}$ KB to a disjunctive datalog program.

DLog [193] is an ABox reasoning algorithm for the $\mathcal{SHIQ}$ language that permits storing the content of the ABox externally in a database and answers instance check and instance retrieval queries by transforming the KB into a Prolog program. TRILL differs from these works for the considered DL and from DLog for the capability of answering general queries.

A different approach is shown in [194], that introduced a system for reasoning on a logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) ASP and can interoperate with OWL. This system, called OntoDLV,

rewrites the OWL KB into the OntoDLP language, can retrieve information directly from external OWL ontologies and answers queries by using ASP. OntoDLV cannot find the set of explanations thus it is not applicable to DISPONTE KBs.

All the above systems are not able to compute the probability of queries. One of the first probabilistic reasoners is PRONTO [195]. This system is based on Pellet like BUNDLE, but differently from it, PRONTO exploits also a linear program solver such as GLPK[13] in order to execute inference on P-$\mathcal{SHIQ}(\mathbf{D})$ [114] KBs. as briefly described in Section 9.3, in these KBs the probabilistic part contains conditional constraints of the form $(D|C)[l, u]$ that informally mean "generally, if an object belongs to C, then it belongs to D with a probability in the interval $[l, u]$". PRONTO performs probabilistic lexicographic entailment by means of solving Probabilistic Satisfiability problems (PSATs) and tight logical entailments. Pellet is used to help the generation of linear programs given as input to the linear program solver.

FOProbLog [196] is an extension of ProbLog where a program contains a set of *probabilistic facts*, i.e. facts annotated with probabilities, and a set of general clauses which can have positive and negative probabilistic facts in their body. Each fact is assumed to be probabilistically independent. FOProbLog follows the distribution semantics and exploits BDDs to compute the probability of queries. FOProbLog is a reasoner for FOL that is not tailored to DLs, so the algorithm could be suboptimal for them.

A combination between DLs and logic programs was presented in [115] in order to integrate ontologies and rules. They use a tightly coupled approach to (probabilistic) disjunctive description logic programs. They define a description logic program as a pair (L, P), where L is a DL KB and P is a disjunctive logic program which contains rules on concepts and roles of L. P may contain probabilistic alternatives in the style of ICL [44]. Interpretations assign a probability to ground atoms, in the style of Nilsson probabilistic logic [197]. Queries can be answered by finding all answer sets. Differently from [115], in DISPONTE interpretations are not probabilistic and they are assigned a probability, instead of being a mapping from atoms to probabilities.

In [119] and [198] a KB is associated with a Bayesian network with variables V. Axioms take the form $E : X = x$ where E is a DL axiom and $X = x$ is an annotation with $X \subseteq V$ and x a set of values for these variables. The Bayesian network assigns a probability to every assignment of V, called a world. The authors show that the probability of a query $Q = E : X = x$ is given by the sum of the probabilities of the worlds where $X = x$ is satisfied and where E is a logical consequence of the theory composed of the axioms whose annotation is true in the world. DISPONTE is a special case of these semantics where every axiom $E_i : X_i = x_i$ is such that X_i is a single Boolean variable and the Bayesian network has no edges, i.e., all the variables are independent. This is an important special case that greatly simplifies reasoning, as computing the probability of the worlds takes a time linear in the number of variables. However, in case the added expressiveness of these formalisms is needed, the Bayesian network could be translated into an equivalent one with only mutually unconditionally independent random variables as shown in Figure 9.2.

A similar approach was presented in [120] where Markov networks are used instead of Bayesian networks. The approach of [198] was applied in [199, 200] to extend the

[13] `https://www.gnu.org/software/glpk/`

$\mathcal{EL}$ DL, defining the probabilistic DL called $\mathcal{BEL}$. The system BORN [200] answers probabilistic subsumption queries w.r.t. $\mathcal{BEL}$ KBs. It exploits ProbLog for managing the probabilistic part of the KB.

In [201] and [202], we addressed representation and reasoning for Datalog$^{\pm}$ ontologies in an Abductive Logic Programming framework, with existential and universal variables, and Constraint Logic Programming constraints in rule heads. The underlying abductive proof procedure can be directly exploited as an ontological reasoner for query answering and consistency check.

13.10 Conclusion

In this chapter we presented three inference systems for probabilistic description logics that follow DISPONTE.

The first one is BUNDLE, a system written in Java to perform probabilistic logical inference on OWL 2 KB. It tries to obtain the covering set of minimal explanations for a given query and then uses BDDs to make the explanations pairwise incompatible. In [185, 3] the authors compared BUNDLE with PRONTO showing that BUNDLE has generally better performances. However, there is still room for improvement in this system. Currently BUNDLE depends on Pellet to get explanations. Unfortunately, this reasoner has become closed source since version 2.3.0 and we are currently making efforts to modularize BUNDLE and make it independent of the reasoner used to get explanations.

The second system is TRILL, which implements the tableau algorithm in Prolog and exploits Prolog's backtracking facility to obtain all the explanations of a given query. TRILL is limited to $\mathcal{SHIQ}(\mathbf{D})$ KBs. In the future we plan to extend TRILL by adding the tableau rules defined in [61] in order to support $\mathcal{SROIQ}(\mathbf{D})$ KBs.

Finally the third one is called TRILL^P. It uses the tableau rules defined in Figure 8.4, but, instead of computing all the explanations for a given query, TRILL^P is based on the approach defined in [101] and computes the pinpointing formula by exploiting a SAT solver to test ψ-*insertability*. Due to this approach TRILL^P is limited to $\mathcal{SHI}$ KBs.

We developed a web application for for TRILL and TRILL^P called TRILL on SWISH (Section 13.6).

BUNDLE was compared in [185, 3] to PRONTO and BUNDLE showed better performances. Starting from these results we compared BUNDLE with TRILL and TRILL^P, the results of our experiments are reported in Section 13.8.

This chapter concludes the part of this thesis related to logical inference on probabilistic logics. In the next part we present (distributed) algorithms and systems for parameter and structure learning.

Part IV

Learning

Chapter 14

Introduction to Statistical Relational Learning

This chapter is devoted to introduce Inductive Logic Programming and its probabilistic evolution **Probabilistic Inductive Logic Programming**, also known as **Statistical Relational Learning**. After a brief introduction in Section 14.1, the basic principles of ILP and SRL are provided in Section 14.2 and Section 14.3 respectively. Section 14.4 draws conclusion.

14.1 Introduction

In **Part II - Probabilistic Logics** of this thesis we have introduced some probabilistic logical formalisms to represent knowledge, while in **Part III - Inference in Probabilistic Logics** we have discussed various methods to make inference with this type of formalisms. This part of the thesis deals with machine learning, i.e. how to automatically learn new knowledge from data. This part of machine learning is mainly related to the Statistical Relational Learning (SRL), a.k.a. Probabilistic Inductive Logic Programming (PILP), a probabilistic evolution of Inductive Logic Programming (ILP).

In addition, in this work we deal with supervised learning (as opposed to unsupervised and reinforcement learning). It means that our learning system is trained with input-output examples. Given this training data, the learning system tries to learn a mapping function between input and output, so that it is applicable with new inputs never seen before.

Inductive Logic Programming (ILP) and its probabilistic evolution **SRL** has some important advantages over other machine learning approaches.

- Logic, and First-Order Logic in particular, is a very well known mathematical field, which provides theoretical foundations to ILP concept, approaches and results.

- Logical formalisms provide expressive and uniform means of representation: the background KB, the examples and the induced hypothesis can all be represented as logic formulas.

- Knowledge represented by a logical formalism is human-readable and understandable. A set of logic formulas induced by an ILP or SRL system is easy to interpret for a human.

However, ILP (and consequently SRL) has some important drawbacks, compared to other machine learning approaches.

- It is generally slower. This is mainly due to the fact that ILP performs combinatorial search and relies on deduction, i.e. reasoning, to find and to prove that an induced hypothesis is correct. Reasoning can be highly computational expensive. The more expressive a logic is, the more complex the reasoning.

- ILP can handle a small number of examples. Learning from small numbers of examples is more difficult and unreliable than learning from lots of data.

ILP is unable to handle uncertain data, for this reason was extended to SRL. However, reasoning in SRL is even more complex than ILP. In fact, in Chapter 11 we explained that to perform exact probabilistic logical inference of a given query, you have to find all the explanations of that query and then make them mutually exclusive.

In the following section we provide an introduction to ILP and SRL, and discuss the main problems they tackle.

14.2 Inductive Logic Programming

ILP has been defined by Muggleton in [203, 204] as an intersection of Machine Learning and Logic Programming. In ILP the goal of the learning process is to find **hypotheses**, in the form of logic program, that provide information about the instance data starting from a set of positive and negative examples (*induction*). In ILP the knowledge is expressed in some logical formalism. ILP traditionally uses logic programs like Prolog for logical representation, but the same principles are applicable to Description Logics (DLs).

The ILP learning problem can be defined as follows.

Definition 14.1 ILP Learning Problem
Given:

- a hypothesis language $\mathcal{L}_H$;
- a set of positive examples $\mathcal{E}^+$;
- a set of negative examples $\mathcal{E}^-$;
- a background KB $\mathcal{K}$; and
- a *covers* relation $covers(e, H, \mathcal{K}) \in \{false, true\}$ that returns the classification of an example e with respect to H and $\mathcal{K}$.

Induce a hypothesis, i.e. a logic theory, $H \in \mathcal{L}_H$ such that H *covers* all positive examples and none of the negative examples. □

The found hypothesis can be **complete**, **consistent** and/or **correct**.

Definition 14.2 Hypothesis Completeness
A hypothesis H is *complete* with respect to the set of positive examples $\mathcal{E}^+$ iff $\forall e \in \mathcal{E}^+$ H covers e. Otherwise it is *too week*. □

Definition 14.3 Hypothesis Consistency
A hypothesis H is *consistent* with respect to the set of negative examples $\mathcal{E}^-$ iff $\forall e \in \mathcal{E}^-$ H does not cover e. Otherwise it is *too strong*. □

Definition 14.4 Hypothesis Correctness
H is *correct* with respect to $\mathcal{E}^+$ and $\mathcal{E}^-$ iff H is complete w.r.t. $\mathcal{E}^+$ and consistent w.r.t. $\mathcal{E}^-$. A hypothesis is *overly general* if it is complete, but not consistent. It is *overly specific* if it is consistent, but not complete. □

Definition 14.5 Generality Relation
A hypothesis G is *more general* than a hypothesis S, denoted $G \succeq S$, if G covers all the instances that are also covered by S, denoted as $covers(S) \subseteq covers(G)$. G is called a *generalization* of S, and S a *specialization* of G. □

In ILP, finding a satisfactory hypothesis means that we have to search the correct theory in the hypothesis space defined by the hypothesis language $\mathcal{L}_H$. Two steps are usually used to find the correct hypothesis: specialization and generalization.

Generalization If the current hypothesis H together with the background KB is not complete, i.e. it does not cover all positive examples, then it means that H is too weak and one needs to find a more general hypothesis such that all positive examples are covered.

Specialization If the current hypothesis together with the background knowledge is not consistent, i.e. some negative examples are covered by H, then it means that H is too strong and one needs to find a more specific hypothesis such that is consistent with respect to the negative examples.

Usually two paradigms are used to search the hypothesis: **top-down** and **bottom-up**. While top-down approaches successively specialize a very general starting hypothesis, bottom-up approaches successively generalize a very specific one.

The search of the correct hypothesis in the hypothesis space $\mathcal{L}_H$ is performed by the *refinement operator*.

Definition 14.6 ILP Refinement Operator
Given a quasi-ordered set $\langle \mathcal{L}_H, \succeq \rangle$, where $\mathcal{L}_H$ is a logical formalism used as hypothesis language and $\succeq$ is the generality relation defined in Definition 14.5. A **refinement operator** is a function

$$\rho : \mathcal{L}_H \rightarrow \mathcal{P}(\mathcal{L}_H)$$

where $\mathcal{P}(\mathcal{L}_H)$ is the powerset of $\mathcal{L}_H$. □

We can have two types of refinement operators: *downward* and *upward*, that can be used, respectively, for specialization and generalization of a hypothesis.

Definition 14.7 Downward Refinement Operator
A **downward refinement operator** for a quasi-ordered set $\langle \mathcal{L}_H, \succeq \rangle$ is a refinement operator ρ such that, for every $H \in \mathcal{L}_H$

$$\rho(H) \subseteq \{H' \mid H \succeq H'\}$$

□

Definition 14.8 Upward Refinement Operator
An **upward refinement operator** for a quasi-ordered set $\langle \mathcal{L}_H, \succeq \rangle$ is a refinement operator ρ such that, for every $H \in \mathcal{L}_H$

$$\rho(H) \subseteq \{H' \mid H \preceq H'\}$$

□

It still remains to define the *coverage* relation. In the literature [205, 206], various ILP settings has been considered. The most notably ones are: **learning from entailment** [207, 208], **learning from interpretations** [209, 210, 5] and **learning from proofs** [211].

Definition 14.9 Learning from Entailment
In the learning from entailment setting, a hypothesis H covers an example e with respect to a background KB $\mathcal{K}$ iff $\mathcal{K} \cup H \models e$. Moreover we say that G is more general that S iff $G \models S$. □

Definition 14.10 Learning from Interpretations
In the learning from interpretations setting, a hypothesis H covers an example e with respect to a background KB $\mathcal{K}$ iff $e \models \mathcal{K} \cup H$. Moreover we say that G is more general that S iff $S \models G$. □

Definition 14.11 Learning from Proofs
In the learning from proofs setting, the examples are logical proofs and an example e is covered by a hypothesis H with respect to a background KB $\mathcal{K}$ iff e is a proof for $\mathcal{K} \cup H$. □

See [212] for details about ILP.

14.3 Statistical Relational Learning

ILP was developed to cope with relational data and it does not handle uncertainty. However, in the real world domain, the information is often uncertain. We want to represent this kind of information and be able to perform reasoning and learning over it. To fulfill this need a new research area known as **Statistical Relational Learning (SRL)** [213, 214], **Probabilistic Inductive Logic Programming (PILP)** [205, 206] or **Probabilistic Logic Learning** [215] extends ILP with probability. SRL combines principles and ideas from three important subfields of Artificial Intelligence: machine learning, knowledge representation and reasoning on uncertainty.

In Part II we introduced the distribution semantics. This semantics provide a powerful mechanism to combine logical representation with the probability theory and led to the definition of several Probabilistic Logic Programming (PLP) languages, such as PRISM [1], ProbLog [45] and LPADs [31] (see Chapter 6). In Chapter 9 the distribution semantics has been applied to Description Logics (DLs) and the resulting semantics was named DISPONTE.

Once the semantics was defined, the general SRL learning problem can be formalized as follows.

Definition 14.12 SRL Learning Problem
Given:

- a probabilistic logical formalism $\mathcal{L}_H$;
- a set of positive examples $\mathcal{E}^+$;
- a set of negative examples $\mathcal{E}^-$;
- a background KB $\mathcal{K}$; and
- a probabilistic coverage relation $covers(e, H, \mathcal{K}) = P(e \mid H, \mathcal{K})$.

Induce a hypothesis $H^* \in \mathcal{L}_H$ that maximise the *maximum likelihood*[1]

$$H^* = \arg\max_H \prod_{e^+ \in \mathcal{E}^+} P(e^+, H, \mathcal{K}) \prod_{e^- \in \mathcal{E}^-} (1 - P(e^-, H, \mathcal{K})) \tag{14.1}$$

or the *maximum log-likelihood*

$$H^* = \arg\max_H \sum_{e^+ \in \mathcal{E}^+} \log P(e^+, H, \mathcal{K}) + \sum_{e^- \in \mathcal{E}^-} \log(1 - P(e^-, H, \mathcal{K})) \tag{14.2}$$

□

If the user wants a correct hypothesis the following constraints should hold:

- $\forall e^+ \in \mathcal{E}^+$, $covers(e^+, H^*, \mathcal{K}) = P(e^+ \mid H^*, \mathcal{K}) > 0$
- $\forall e^- \in \mathcal{E}^-$, $covers(e^-, H^*, \mathcal{K}) = P(e^- \mid H^*, \mathcal{K}) = 0$

The hypothesis is essentially a probabilistic logic KB $\mathcal{S}$ annotated with probabilistic parameters λ, i.e. $H = (\mathcal{S}, \lambda)$. In SRL we have typically two kinds of learning problems:

Parameter learning The underlying logic KB $\mathcal{S}$, also called *structure*, is assumed to be fixed, and the learning task consists of estimating the parameters λ that maximize the likelihood.

Structure learning Both $\mathcal{S}$ and λ have to be learned.

[1] Assuming that the examples are independent and identically distributed (i.i.d.).

14.3.1 Parameter Learning

The problem of parameter learning concerns the estimation of the values of the parameters λ of a fixed probabilistic logic KB that best explain the examples $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$. The parameter learning problem can be formalized as follows.

Definition 14.13
Given

- a set of positive examples $\mathcal{E}^+$;
- a set of negative examples $\mathcal{E}^-$;
- a probabilistic model $M(\lambda) = (\mathcal{S}, \lambda)$, with structure $\mathcal{S}$, i.e. a fixed probabilistic KB, and parameters λ;
- a probabilistic coverage relation $covers(e, H, \mathcal{K}) = P(e \mid M(\lambda))$ which computes the probability of the example e given the model $M(\lambda)$;
- a scoring function $score(\mathcal{E}, M(\lambda))$ which uses the probabilistic coverage relation.

Induce the parameters λ^* such that

$$\lambda^* = \arg\max_{\lambda} score(\mathcal{E}, M(\lambda)) \tag{14.3}$$

□

If all examples are fully observable, maximum (log-)likelihood reduces to frequency counting. If instead we are dealing with missing data, i.e. with partially observed data, we cannot compute the maximum likelihood estimates with closed form formulas. It is a numerical optimization problem. One of the most commonly adapted techniques for SRL is the Expectation-Maximization (EM) algorithm [216]. This algorithm randomly initializes the parameters and then iteratively performs the following two steps until convergence:

Expectation Step Given the partially observed data and the present parameters of the model, estimate the conditional distribution of the unobserved variables, also known as *hidden variables* or *latent variables*.

Maximization Step Update the parameters of the structure $\mathcal{S}$ by using the expectations computed in the previous step.

EM is exploited in many systems such as PRISM [217], LFI-ProbLog [30], EMBLEM [4] and RIB [218].

Gradient descent methods compute the gradient of the target function and iteratively modify the parameters following the direction of the gradient. LeProbLog [219] is a system that uses a dynamic programming algorithm for computing the gradient exploiting BDDs.

14.3.2 Structure Learning

In the parameter learning problem we have assumed that the structure of the probabilistic model is given and fixed. This is not always the case. In structure learning, the problem is to learn both the structure and the parameters of an initial probabilistic logical KB. The problem can be formalized as follows.

Definition 14.14
Given

- a set of positive examples $\mathcal{E}^+$;
- a set of negative examples $\mathcal{E}^-$;
- a language $\mathcal{L}_M$ of possible models of the form $M = (\mathcal{S}, \lambda)$, with structure $\mathcal{S}$, i.e. a probabilistic KB, and parameters λ;
- a probabilistic coverage relation $covers(e, H, \mathcal{K}) = P(e \mid M)$ which computes the probability of the example e given the model M;
- a scoring function $score(\mathcal{E}, M)$ which uses the probabilistic coverage relation.

Induce the model $M^* = (\mathcal{S}, \lambda)$ such that

$$M^* = \arg\max_M score(\mathcal{E}, M) \tag{14.4}$$

□

Like the ILP problem, structure learning is essentially a search problem. There is a space of possible models, defined by $\mathcal{L}_M$, to be traversed. The goal is to find the best one according to the scoring function.

See [213] and [214] for an introduction to SRL.

14.4 Conclusion

In this chapter we introduced Inductive Logic Programming (ILP) and its evolution Statistical Relational Learning (SRL). Moreover we presented the problems that these research fields try to tackle. In particular, we provided a formalization of the parameter learning problem, that concerns the estimation of the probabilistic parameters of a given probabilistic logic KB, and a formalization of the structure learning problem, where we are interested in inferring both the structure and the parameters of a probabilistic logic KB.

The next chapters will present and discuss algorithms for parameter and structure learning on LPADs and PDLs that follow DISPONTE. The execution of these algorithms is rather expensive from a computational point of view, taking a few hours on datasets of the order of MBs. Therefore, in order to efficiently manage larger datasets, we distributed these algorithms using a MapReduce approach.

Chapter 15

Distributed Learning in Probabilistic Logic Programming

In this chapter we illustrate some algorithms for (distributed) parameter and structure learning. The chapter is organized as follows. After a brief introduction of the considered problems and the state of the art in Section 15.1, Section 15.2 and Section 15.3 briefly describe EMBLEM and SLIPCOVER, two algorithms for parameter and structure learning respectively. EMBLEM$^{\text{MR}}$, a distributed version of EMBLEM, is illustrated in Section 15.4, whereas in Section 15.5 presents SEMPRE, the distributed version of SLIPCOVER.

15.1 Introduction

In Chapter 14 we said that the two main tasks of SRL are parameter and structure learning. LPADs [31] discussed in Section 6.2 are a probabilistic logical formalisms for the PLP framework. The parameter learning problem was considered since the birth of the distribution semantics [1]. In fact in [1], Taisuke Sato, besides the distribution semantics, proposed a parameter learning algorithm based on the EM algorithm. Starting from the work of Ishihata et al. [220] the authors of [4] developed EMBLEM for parameter learning on LPADs. Other algorithms have been proposed for learning the parameters of probabilistic logic programs under the distribution semantics, such as PRISM [217] and ProbLog2 [30]. Recently, systems for learning the structure of these programs have started to appear. Among these, SLIPCOVER [5] which exploits EMBLEM for structure learning on LPADs.

EMBLEM and SLIPCOVER are able to learn good quality solutions in a variety of domains [5] but they are usually computational expensive, often taking some hours to complete on datasets of the order of MBs. In order to deal with Big Data, it is fundamental to reduce learning times by exploiting modern computing infrastructures such as clusters and clouds.

Therefore, distributed version of EMBLEM and SLIPCOVER were developed, namely EMBLEM$^{\text{MR}}$ and SEMPRE, which both exploit the MapReduce approach. MapReduce [221] distributes the work among a pool of "mapper" and "reducer" workers. The computation is performed by dividing the input among the mappers, each taking a set of units of information and returning a set of (key, value) pairs. These sets

are then given to reducers in the form of pairs (key, list of values) and the reducers compute an aggregate of the values returning a set of (key, aggregated value) couples that represent the output of the task.

15.2 Parameter Learning: EMBLEM

EMBLEM [4], for "EM over Bdds for probabilistic Logic programs Efficient Mining" performs parameter learning of LPADs by using an Expectation Maximization (EM) algorithm (see Algorithm 15.1). It takes as input a set of interpretations I and the theory $\mathcal{T}$ for which we want to learn the parameters. An interpretation is a set of ground facts describing a portion of the domain. EMBLEM is targeted at discriminative learning, since the user has to indicate which predicate(s) of the domain is/are *target*, the one(s) for which we are interested in good predictions. The interpretations must contain also negative facts for target predicates. The ground atoms for the target predicates (E) represent the positive and negative examples (*queries*) for which BDDs are built, encoding the disjunction of their explanations.

Algorithm 15.1 EMBLEM algorithm.

```
1: function EMBLEM(I, T, ϵ, δ)
2:     Identify examples E
3:     Build BDDs for the examples E using T and I
4:     LL = −∞
5:     repeat
6:         LL_0 = LL
7:         LL = Expectation(BDDs)
8:         Maximization
9:     until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
10:    return LL, π
11: end function
```

After building the BDDs, EMBLEM maximizes the LL for the positive and negative target examples with an EM cycle, until it has reached a local maximum. The E-step computes the expectations of the latent variables directly over BDDs and returns the LL of the data that is used in the stopping criterion. The expected counts are then used in the M-step, which updates the parameters π for all clauses for the next EM iteration by relative frequency.

For each target fact Q, the expectations are $\mathbf{E}[X_{ijk} = x|Q]$ for all C_is, $k = 1, \ldots, n_i - 1$, $j \in g(i) := \{j|\theta_j$ is a substitution grounding $C_i\}$ and $x \in \{0, 1\}$. $\mathbf{E}[X_{ijk} = x|Q]$ is given by

$$\mathbf{E}[X_{ijk} = x|Q] = P(X_{ijk} = x|Q) \cdot 1 + P(X_{ijk} = (1 - x)|Q) \cdot 0 = P(X_{ijk} = x|Q).$$

From $\mathbf{E}[X_{ijk} = x|Q]$ one can compute the expectations $\mathbf{E}[c_{ik0}|Q]$ and $\mathbf{E}[c_{ik1}|Q]$ where c_{ikx} is the number of times a Boolean variable X_{ijk} takes on value x for $x \in \{0, 1\}$ and for all $j \in g(i)$:

$$\mathbf{E}[c_{ikx}|Q] = \sum_{j \in g(i)} \mathbf{E}[X_{ijk} = x|Q].$$

The expected counts $\mathbf{E}[c_{ik0}]$ and $\mathbf{E}[c_{ik1}]$ are obtained by summing $\mathbf{E}[c_{ik0}|Q]$ and $\mathbf{E}[c_{ik1}|Q]$ over all examples:

$$\mathbf{E}[c_{ikx}] = \sum_{Q} \mathbf{E}[c_{ikx}|Q].$$

$P(X_{ijk} = x|Q)$ is given by $\frac{P(X_{ijk}=x,Q)}{P(Q)}$, where $P(X_{ijk} = x, Q)$ and $P(Q)$ can be computed with two traversals of the BDD built for the query Q.

EMBLEM is strictly related to EDGE, the two algorithms are very similar. The latter is a parameter learner for PDLs and is discussed in detail in Chapter 16, whereas we refer to [4] for details about the former.

15.3 Structure Learning: SLIPCOVER

SLIPCOVER [5] (see Algorithm 15.2) learns the structure of probabilistic logic programs with a two-phase search strategy: (1) beam search in the space of clauses in order to find a set of promising clauses and (2) greedy search in the space of theories. In the first phase, SLIPCOVER performs clause search for each target predicate separately. The beam for each target predicate is initialized (Function InitialBeams) with a number of bottom clauses built as in Progol [222]. Then SLIPCOVER generates refinements of the best clause in the beam and evaluates them through LL by invoking EMBLEM. Each clause is then inserted in the new beam of promising clauses and in the sets of target and background clauses ordered according to the LL. This is repeated until the original beam becomes empty. The whole process is repeated at most NI steps.

The search in the space of theories starts from an empty theory which is iteratively extended with one target clause at a time from those generated in the previous beam search. The algorithm starts with an empty theory and then iteratively adds a new clause to the theory, runs EMBLEM to compute the corresponding LL and checks whether to keep the clause in the theory or not. If the LL of the new theory decreases, SLIPCOVER discards the clause.

Finally, background clauses, the ones with a non-target predicate in the head, are added en bloc to the theory so built. A further parameter optimization step is executed with EMBLEM and clauses that are never involved in an example derivation are removed.

Algorithm 15.2 Function SLIPCOVER

```
1: function SLIPCOVER(I, NInt, NS, NA, NI, NV, NB, NTC, NBC, ϵ, δ)
2:     IBs =InitialBeams(I, NInt, NS, NA)                          ▷ Clause search
3:     TC ← []
4:     BC ← []
5:     for all (PredSpec, Beam) ∈ IBs do
6:         Steps ← 1
7:         NewBeam ← []
8:         repeat
9:             while Beam is not empty do
10:                Remove the first couple ((Cl, Literals), LL) from Beam   ▷ Remove the first clause
11:                Refs ←ClauseRefinements((Cl, Literals), NV) ▷ Find all refinements Refs of (Cl, Literals) with at most NV variables
12:                for all (Cl′, Literals′) ∈ Refs do
13:                    (LL″, {Cl″}) ←EMBLEM(I, {Cl′}, ϵ, δ)
14:                    NewBeam ←Insert((Cl″, Literals′), LL″, NewBeam, NB)
15:                    if Cl″ is range restricted then
16:                        if Cl″ has a target predicate in the head then
17:                            TC ←Insert((Cl″, Literals′), LL″, TC, NTC)
18:                        else
19:                            BC ←Insert((Cl″, Literals′), LL″, BC, NBC)
20:                        end if
21:                    end if
22:                end for
23:            end while
24:            Beam ← NewBeam
25:            Steps ← Steps + 1
26:        until Steps > NI
27:    end for
28:    T ← ∅, TLL ← −∞                                             ▷ Theory search
29:    repeat
30:        Remove the first couple (Cl, LL) from TC
31:        (LL′, T′) ←EMBLEM(I, T ∪ {Cl}, ϵ, δ)
32:        if LL′ > TLL then
33:            T ← T′, TLL ← LL′
34:        end if
35:    until TC is empty
36:    T ← T ⋃(Cl,LL)∈BC {Cl}
37:    (LL, T) ←EMBLEM(I, T, D, NEM, ϵ, δ)
38:    return T
39: end function
```

15.4 Distributed Parameter Learning: EMBLEM$^{\mathrm{MR}}$

In order to parallelize structure learning, first we developed a MapReduce version of EMBLEM called EMBLEM$^{\mathrm{MR}}$, where the Expectation step is performed in parallel following the approach proposed in [223] for applying MapReduce to the EM algorithm.

In particular, EMBLEM$^{\mathrm{MR}}$ (see Algorithm 15.3) creates n workers indexed from 1 to n. Worker 1 is the "master" and is in charge of splitting work among the "slaves" (the other $n-1$ workers). The Map function is performed by all processes; the Reduce function and the Maximization step are performed by the master (also referred to as the "reducer").

During the Map phase, the input interpretations I and the input theory $\mathcal{T}$ whose parameters are to be learned are replicated among all workers, while the examples E are evenly divided into n subsets $E_1, \ldots, E_n$ by the master. When splitting examples, E_1 is handled by the master, while $E_2, \ldots, E_n$ are sent to the slaves (also referred to as "mappers") (line 4-5). The m-th subset is sent to mapper m that builds the BDDs for the examples belonging to it (line 6 and lines 18-19). The assignment of subsets of examples to different mappers is possible because each of them stores I and $\mathcal{T}$ in

main memory and each example (and thus each BDD) is independent of the others, allowing one to divide and treat them separately. After that, all the mappers stay active keeping the BDDs in memory.

During the learning phase (EM cycle), the Expectation step is executed in parallel. The master sends the current values of the parameters to each mapper m and computes the expectations for its examples (lines 10-11). The slaves, after receiving the parameters π, compute the expectations for their subset of examples, using the BDDs stored in memory (lines 22-23). By keeping the BDDs in memory, the mappers only need to receive the parameters' updated values to accomplish their task.

Then, during the Reduce phase, the expectations are sent back to the reducer (line 24), that simply sums them up with its own values obtaining the total expected counts (lines 12-13). Finally, the Maximization step is performed serially (line 14).

This parallelization strategy is implemented using the Message Passing Interface (MPI). We preferred it over a standard MapReduce framework (such as Hadoop) because we wanted to customize the parallelization strategy to better suit our needs: our mappers have side-effects because they have to retain in main memory all the BDDs through all iterations, so they are not purely functional, as is required by standard MapReduce frameworks. This would have been a limitation because it would have forced us to build the BDDs in every step.

Algorithm 15.3 Function EMBLEM$^{\text{MR}}$

```
 1: function EMBLEM^MR(I, T, n, ϵ, δ)
 2:     if MASTER then
 3:         Identify examples E
 4:         Split examples E into n subsets E_1, ..., E_n
 5:         Send E_m to each worker m, 2 ≤ m ≤ n
 6:         Build BDDs_1 for examples E_1 using T and I
 7:         LL = −∞
 8:         repeat
 9:             LL_0 = LL
10:             Send the parameters π to each worker m, 2 ≤ m ≤ n
11:             LL = EXPECTATION(BDDs_1)
12:             Collect LL_m and the expectations from each worker m, 2 ≤ m ≤ n
13:             Update LL and the expectations
14:             MAXIMIZATION
15:         until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
16:         return LL, π
17:     else                                                    ▷ the j-th slave
18:         Receive E_j from master
19:         Build BDDs_j for examples E_j using T and I
20:         LL = −∞
21:         repeat
22:             Receive the parameters π from master
23:             LL_j = EXPECTATION(BDDs_j)
24:             Send LL_j and the expectations to master
25:         until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
26:     end if
27: end function
```

15.5 Distributed Structure Learning: SEMPRE

SEMPRE (see Algorithm 15.4) parallelizes three operations of the structure learning algorithm SLIPCOVER by employing n workers, one master and $n-1$ slaves. All the workers initially receive all the input data.

The first operation is the scoring of the clause refinements: when the revisions *Refs* for a clause are generated (line 12), the master process splits them evenly into n subsets $Refs_1, \ldots, Refs_n$ and assigns $Refs_2, ..., Refs_n$ to the slaves. The subset $Refs_1$ is handled by the master. Then, SEMPRE enters the *Map phase* (lines 20-30), when each worker is listening for requests to score a set of refinements and returns the set of refinements with their log-likelihood (LL). Scoring is performed using (serial) EMBLEM which is run over a theory containing only one clause: since the BDDs built for clauses are usually small, using EMBLEM$^{\text{MR}}$ would imply a too large overhead.

Once the master has received all sets of scored refinements from the workers, it enters the *Reduce phase* (lines 32-35), where it updates the beam of promising clauses (*NewBeam*) and the sets of target and background clauses (TC and BC respectively): the scored refinements are inserted in order of LL into these lists. NTC (NBC) is the maximum size for TC (BC).

The second parallelized operation is parameter learning for the theories. In this phase (lines 45-52), each clause from TC is tentatively added to the theory, which is initially empty. In the end, it contains all the clauses that improved its LL (search in the space of theories). In this case, the BDDs that are being built can be quite complex since the theory contains multiple clauses, so EMBLEM$^{\text{MR}}$ is used.

The third parallelized operation is the final parameter optimization for the theory including also the background clauses (lines 53-54). All the background clauses are added to the theory previously learned and the parameters of the theory are learned by means of EMBLEM$^{\text{MR}}$ because of the computational complexity.

Algorithm 15.4 Function SEMPRE

```
1: function SEMPRE(I, n, NInt, NS, NA, NI, NV, NB, NTC, NBC, ϵ, δ)
2:     IBs =InitialBeams(I, NInt, NS, NA)                                  ▷ Clause search
3:     TC ← []
4:     BC ← []
5:     for all (PredSpec, Beam) ∈ IBs do
6:         Steps ← 1
7:         NewBeam ← []
8:         repeat
9:             while Beam is not empty do
10:                if MASTER then
11:                    Remove the first couple ((Cl, Literals), LL) from Beam        ▷ Remove the first clause
12:                    Refs ←ClauseRefinements((Cl, Literals), NV)   ▷ Find all refinements Refs of (Cl, Literals)
   with at most NV variables
13:                    Split evenly Refs into n subsets Refs_1, ..., Refs_n
14:                    for m = 2 to n do
15:                        Send Refs_m to worker m
16:                    end for
17:                else                                                        ▷ the j-th slave
18:                    Receive Refs_j from master
19:                end if
20:                for all (Cl', Literals') ∈ Refs_j do
21:                    (LL'', {Cl''}) ←EMBLEM(I, {Cl'}, ϵ, δ)
22:                    NewBeam ←Insert((Cl'', Literals'), LL'', NewBeam, NB)
23:                    if Cl'' is range restricted then
24:                        if Cl'' has a target predicate in the head then
25:                            TC ←Insert((Cl'', Literals'), LL'', TC, NTC)
26:                        else
27:                            BC ←Insert((Cl'', Literals'), LL'', BC, NBC)
28:                        end if
29:                    end if
30:                end for
31:                if MASTER then
32:                    for m = 2 to n do
33:                        Collect the set {(LL'', {Cl''})|∀(Cl', Literals) ∈ Refs_m} from worker m
34:                        Update NewBeam, TC, BC
35:                    end for
36:                else                                                        ▷ the j-th slave
37:                    Send the set {(LL'', {Cl''})|∀(Cl', Literals) ∈ Refs_j} to master
38:                end if
39:            end while
40:            Beam ← NewBeam
41:            Steps ← Steps + 1
42:        until Steps > NI
43:    end for
44:    if MASTER then
45:        T ← ∅, TLL ← −∞                                                     ▷ Theory search
46:        repeat
47:            Remove the first couple (Cl, LL) from TC
48:            (LL', T') ←EMBLEM^MR(I, T ∪ {Cl}, n, ϵ, δ)
49:            if LL' > TLL then
50:                T ← T', TLL ← LL'
51:            end if
52:        until TC is empty
53:        T ← T ⋃_{(Cl,LL)∈BC} {Cl}
54:        (LL, T) ←EMBLEM^MR(I, T, n, ϵ, δ)
55:        return T
56:    end if
57: end function
```

15.6 Experiments

SEMPRE was implemented in Yap Prolog [151] using the `lam_mpi` library for interfacing Prolog with the underlying MPI framework. `lam_mpi` is a built-in library of Yap Prolog which provides an interface to LAM MPI, one of the first implementation of MPI. LAM MPI is now at the basis of the OpenMPI library, one of the most widespread libraries for developing MPI applications.

SEMPRE was tested on the following seven real world datasets: Hepatitis [224], Mutagenesis [225], UWCSE [226], Carcinogenesis [227], IMDB [228], HIV [229] and WebKB [230]. All experiments were performed on GNU/Linux machines with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Table 15.1 shows the wall time in seconds taken by SEMPRE to perform learning averaged over the folds (ten for Mutagenesis, four for WebKB, one for Carcinogenesis and five for all the others). The experiments were performed with 1, 8, 16 or 32 workers. Figure 15.1 shows the speedup of SEMPRE as a function of the number of workers. The speedup for n workers is the fraction of the time for 1 worker over the time for n workers. Ideally, one wants to achieve a linear speedup. The speedup is always larger than 1 and grows with the number of workers achieving the best with 32 workers, except for HIV and IMDB, where there is a slight decrease for 16 and 32 workers due to the overhead caused by the distribution itself; however, these two datasets were the smallest and less in need of a parallel solution.

We have evaluated SEMPRE speedup during both distributed parameter and structure learning. We discovered that it is remarkable in both phases and that SEMPRE spends most of the time in the beam search of clause refinements: for example, for UWCSE the time for clause search is around 94% of the total time, while for WebKB it is around 96%. The average time to handle each refinement is small, around 23ms for UWCSE and 80ms for WebKB. Therefore, the parallelization decisions taken seem justified: since the refinement handling time is small, it does not make sense to perform distributed parameter learning for clause refinements, while it is more reasonable to distribute the refinements to workers. These results show that SEMPRE is able to exploit the availability of processors in most cases.

In order to compare the performances of SEMPRE with other probabilistic learning

Table 15.1: SEMPRE execution time (in seconds) as the number of workers varies.

Dataset	**Number of workers**			
	1	**8**	**16**	**32**
Hepatitis	19,867	4,246	2,392	1,269
Mutagenesis	14,784	2,887	2,587	1,579
UWCSE	12,758	5,401	3,152	1,899
Carcinogenesis	170	23	18	16
IMDB	481	104	113	177
HIV	508	118	136	295
WebKB	2,441	486	322	256

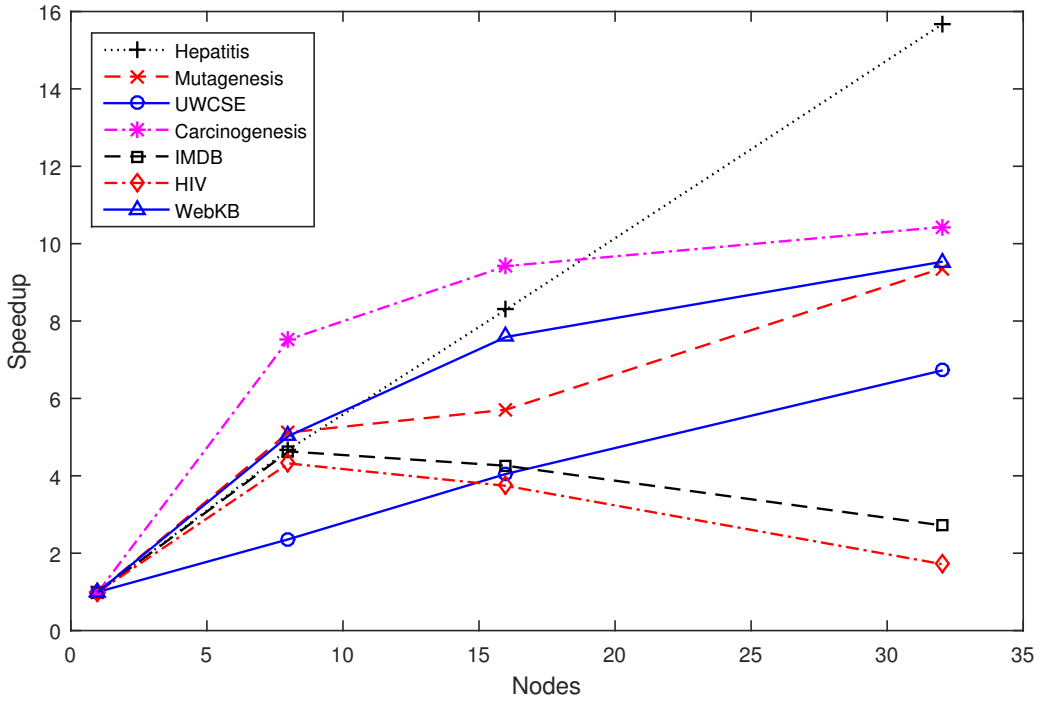


Figure 15.1: SEMPRE speedup graph referred to Table 15.1.

systems we exploited the results presented in [231]. For our purpose we focus on the LEMUR, SLIPCASE [232], ALEPH++ExactL1 [233], MLN-BC [234], MLN-BT [234] and RDN-B [235] systems and on the Mutagenesis, Carcinogenesis, IMDB and HIV datasets. These experiments were performed on GNU/Linux machines with an Intel Core 2 Duo E6550 (2.333 GHz) processor and 4 GB of RAM. In order to compare the learning times of these systems with those of SEMPRE, the execution times of the latter have been scaled of a factor 1,029, which corresponds to the ratio between the two CPU core frequencies.

Tables 15.2, 15.3, 15.4 and 15.5 show the average AUC-PR and time for Mutagenesis, Carcinogenesis, IMDB and HIV respectively. The systems are sorted in decreasing AUC-PR. The experiments show that SEMPRE achieves good quality results, comparable with those of the other systems considered. SEMPRE with multiple workers is often faster than the competing systems, indicating that it can be a valid alternative when execution time is an important factor.

Table 15.2: Results of the experiments in terms of average AUC-PR and execution time (in seconds) on the Mutagenesis dataset. The standard deviations are also shown.

System	AUC-PR	Time(s)
RDN-B	0.964 ± 0.026	70
LEMUR	0.952 ± 0.062	11,230
ALEPH++ExactL1	0.949 ± 0.043	198
SEMPRE 1W	0.948 ± 0.030	15,213
SEMPRE 8W	0.948 ± 0.030	2,971
SEMPRE 16W	0.948 ± 0.030	2,662
SEMPRE 32W	0.948 ± 0.030	1,625
MLN-BT	0.922 ± 0.087	175
SLIPCASE	0.921 ± 0.087	5,135
MLN-BC	0.690 ± 0.201	55

Table 15.3: Results of the experiments in terms of AUC-PR and execution time (in seconds) on the Carcinogenesis dataset.

System	AUC-PR	Time(s)
ALEPH++ExactL1	0.738	147
SEMPRE 1W	0.724	175
SEMPRE 8W	0.724	24
SEMPRE 16W	0.724	19
SEMPRE 32W	0.724	16
LEMUR	0.691	23436
SLIPCASE	0.628	1
MLN-BC	0.619	45
RDN-B	0.550	84
MLN-BT	0.503	114

Table 15.4: Results of the experiments in terms of average AUC-PR and execution time (in seconds) on the IMDB dataset. The standard deviations are also shown.

System	AUC-PR	Time(s)
SEMPRE 1W	1.000 ± 0.000	495
SEMPRE 8W	1.000 ± 0.000	107
SEMPRE 16W	1.000 ± 0.000	116
SEMPRE 32W	1.000 ± 0.000	182
ALEPH++ExactL1	1.000 ± 0.000	9
RDN-B	1.000 ± 0.000	199
SLIPCASE	1.000 ± 0.000	64
LEMUR	1.000 ± 0.000	1,781
MLN-BT	0.977 ± 0.047	459
MLN-BC	0.942 ± 0.071	266

Table 15.5: Results of the experiments in terms of average AUC-PR and execution time (in seconds) on the HIV dataset. The standard deviations are also shown.

System	AUC-PR	Time(s)
SEMPRE 1W	0.844 ± 0.034	523
SEMPRE 8W	0.844 ± 0.034	121
SEMPRE 16W	0.844 ± 0.034	139
SEMPRE 32W	0.844 ± 0.034	304
LEMUR	0.830 ± 0.050	1,290
SLIPCASE	0.777 ± 0.047	44
MLN-BC	0.512 ± 0.041	125
MLN-BT	0.288 ± 0.037	278
RDN-B	0.284 ± 0.057	69

15.7 Conclusions

The main purpose of this chapter was to illustrate the state of the art of distributed learning algorithms for PLP. EMBLEM and SLIPCOVER are two PLP learning algorithms for parameter and structure learning respectively. Starting from this two algorithms we developed their distributed versions: EMBLEM$^{\text{MR}}$ and SEMPRE. These systems exploit the MapReduce approach for performing learning in parallel. SEMPRE has been tested on a number of domains and compared with six different learning systems. The results show that parallelization is indeed effective at reducing running time, even if in some cases the overhead may be significant. Finally, SEMPRE proves to be even more competitive than SLIPCOVER in comparison to the other learning systems, since it inherits SLIPCOVER's performance (in terms of AUCPR) and, in addition, it gains the capability to scale thanks to parallelization.

In the next chapters we presents learning algorithms for PDLs that follow DISPONTE.

Chapter 16

Parameter Learning in Probabilistic Description Logics

This chapter presents EDGE, an algorithm for parameter learning on PDLs. The chapter is organized as follows. In Section 16.1 we provide an introduction, Section 16.2 illustrates the details of EDGE and finally Section 16.3 concludes the chapter.

16.1 Introduction

As mentioned in Section 14.3 one of the main SRL learning problems is parameter learning. In Chapter 15 we discussed some parameter and learning methods for PLP. In particular, in Section 15.2 we illustrated EMBLEM, a parameter learning algorithm for LPADs. In this chapter we present EDGE [14] a parameter learning approach for PDLs that follow DISPONTE.

EDGE [14] adapts the algorithm EMBLEM [4, 236] to the case of PDLs under DISPONTE and exploits the theory presented in [237] It starts from examples of instances (positive examples) and non-instances (negative examples) of concepts and builds BDDs for representing their explanations from the theory. The parameters are then tuned using an EM algorithm [216] in which the required expectations are computed directly on the BDDs.

16.2 EDGE

EDGE, for "Em over bDds for description loGics paramEter learning", takes as input a DL KB and a number of examples that represent the queries. For each query, it generates the BDD encoding its explanations using BUNDLE (see Section 13.2). Usually, the queries are concept assertions divided into *positive* and *negative examples*: positive examples represent information that is true and for which we would like to get high probability, while negative examples are information that we regard as false and for which we would like to get low probability.

After all the BDDs have been encoded, EDGE enters the EM cycle in which expectation and maximization are repeated until the log-likelihood LL of the examples reaches a guaranteed local maximum, which however may not be a global maximum.

The steps of the EM cycle are:

Expectation for each query Q, EDGE computes $\mathbf{E}[c_{i0}|Q]$ and $\mathbf{E}[c_{i1}|Q]$ for all axioms E_i, where c_{ix} is the number of times a variable X_i takes value x, for $x \in \{0, 1\}$:

$$\mathbf{E}[c_{ix}|Q] = P(X_i = x|Q)$$

Then it sums up the contributions of the different examples

$$\mathbf{E}[c_{ix}] = \sum_Q \mathbf{E}[c_{ix}|Q] = \sum_Q P(X_i = x|Q) \tag{16.1}$$

Maximization EDGE computes p_i for all axioms E_i:

$$p_i = \frac{\mathbf{E}[c_{i1}]}{\mathbf{E}[c_{i0}] + \mathbf{E}[c_{i1}]}$$

16.2.1 Expectation Computation

$P(X_i = x|Q)$ is given by $\frac{P(X_i=x,Q)}{P(Q)}$. So (16.1) becomes:

$$\mathbf{E}[c_{ix}] = \sum_Q \frac{P(X_i = x, Q)}{P(Q)} \tag{16.2}$$

Suppose for the moment that we are working with a subset of the family of BDDs called Complete Binary Decision Diagram (CBDD). A CBDD is such that each path from the root to the leaves contains one node for every variable. In particular, on this type of BDDs, the deletion rule is not applied[1]. Then

$$\begin{aligned}
P(X_i = x, Q) &= \sum_{w \in \mathcal{W}_\mathcal{K}} P(X_i = x, Q, w) && (16.3)\\
&= \sum_{w \in \mathcal{W}_\mathcal{K}} P(X_i = x, Q|w)P(w) && (16.4)\\
&= \sum_{w \in \mathcal{W}_\mathcal{K}} P(Q|w)P(X_i = x|w)P(w) && (16.5)\\
&= \sum_{w \in \mathcal{W}_\mathcal{K}: w \models Q} P(X_i = x|w)P(w) && (16.6)\\
& && (16.7)
\end{aligned}$$

where (16.3) and (16.4) follow for the marginalization and the product rule of the theory of probability respectively, (16.5) holds because X_i and Q are conditional independent w.r.t. w and (16.6) stands on because $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. $P(X_i = x|w) = 1$ if $(E_i, x) \in w$ and 0 otherwise.

There is a one to one correspondence between the possible worlds where Q is true and the paths to a 1 leaf in a CBDD. For this reason,

$$P(X_i = x, Q) = \sum_{\rho \in R(Q)} P(X_i = x|\rho) \prod_{d \in \rho} p(d) \tag{16.8}$$

[1]Deletion is performed when both arcs from a node point to the same node (see Definition 10.2).

where w corresponds to a path ρ, $P(X_i = x|w) = P(X_i = x|\rho)$, $R(Q)$ is the set of paths in the BDD for query Q that lead to a 1 leaf, d is an edge of ρ and $p(d)$ is the probability associated with the edge: if d is the 1-branch from a node associated with a variable X_i, then $p(d) = p_i$, if d is the 0-branch, then $p(d) = 1 - p_i$.

Example 16.2.1
Consider the BDD of Example 11.5.3 in Figure 11.4, the equivalent CBDD is shown in Figure 16.1.

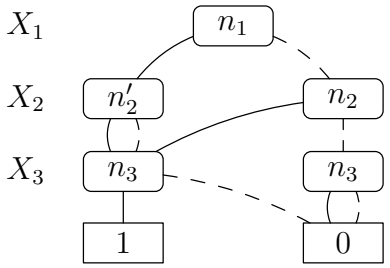


Figure 16.1: Complete Binary Decision Diagram equivalent to the BDD in Example 11.5.3

For a CBDD, $P(X_i = x, Q)$ can be further expanded as

$$P(X_i = x, Q) = \sum_{\rho \in R(Q) \wedge (X_i = x) \in \rho} \prod_{d \in \rho} p(d)$$

where $(X_i = x) \in \rho$ means that ρ contains an x-edge from a node associated with X_i. We can then write

$$P(X_i = x, Q) = \sum_{n \in N(Q) \wedge v(n) = X_i \wedge \rho_n \in R_n(Q) \wedge \rho^n \in R^{child_x(n)}(Q)} p_{ix} \prod_{d \in \rho^n} p(d) \prod_{d \in \rho_n} p(d)$$

where $N(Q)$ is the set of nodes of the CBDD, $v(n)$ is the variable associated with node n, $R_n(Q)$ is the set containing the paths from the root to n, $R^n(Q)$ is the set of paths from n to the 1 leaf and where p_{ix} is p_i if $x = 1$ and $(1 - p_i)$ if $x = 0$. Therefore

$$\begin{aligned} P(X_i = x, Q) &= \sum_{n \in N(Q) \wedge v(n) = X_i} \sum_{\rho_n \in R_n(Q)} p_{ix} \prod_{d \in \rho_n} p(d) \sum_{\rho^n \in R^{child_x(n)}(Q)} \prod_{d \in \rho^n} p(d) \\ &= \sum_{n \in N(Q) \wedge v(n) = X_i} F(n) B(child_x(n)) p_{ix} \end{aligned}$$

where

$$F(n) = \sum_{\rho_n \in R_n(Q)} \prod_{d \in \rho_n} p(d)$$

is called *forward probability* [237], the probability mass of the paths from the root to n, while

$$B(n) = \sum_{\rho^n \in R^n(Q)} \prod_{d \in \rho^n} p(d)$$

is called *backward probability* [237], the probability mass of paths from n to the 1 leaf. If *root* is the root of a BDD for a query Q then $B(root) = P(Q)$.

The expression $F(n)B(child_x(n))p_{ix}$ stands for the sum of the probabilities of all the paths passing through the x-edge of node n. We use the notation $e^x(n)$ to indicate such an expression. Thus

$$P(X_i = x, Q) = \sum_{n \in N(Q) \wedge v(n) = X_i} e^x(n) \tag{16.9}$$

For the case of a fully simplified BDD, i.e. a BDD obtained by applying also the deletion rule, Formula 16.9 is no longer valid since also paths where there is no node associated with X_i can contribute to $P(X_i = x, Q)$. In fact, it is necessary to consider the deleted paths too. Suppose that a node n associated with variable Y has a level higher than variable X_i and suppose that $child_0(n)$ is associated with variable W that has a lower level than variable X_i. The nodes associated with variable X_i have been deleted from the paths from n to $child_0(n)$. If we associate a node m to variable X_i, we can imagine that the current BDD has been obtained from a BDD having a node m that is a descendant of n along the 0-branch and whose outgoing edges both end to $child_0(n)$. The original BDD can be reobtained by applying a deletion operation that merges the two paths passing through m. The probability mass of the two paths that were merged was $e^0(n)(1 - p_i)$ and $e^0(n)p_i$ for the paths passing through the 0-child and 1-child of m respectively.

Let $Del^x(X)$ be the set of nodes n such that the level of X is below that of n and is above the level of $child_x(n)$, i.e., X is deleted between n and $child_x(n)$. For the BDD in Figure 11.4, for example, $Del^1(X_2) = \{n_1\}$, $Del^0(X_2) = \emptyset$, $Del^1(X_3) = \emptyset$, $Del^0(X_3) = \{n_2\}$. Then

$$\begin{aligned} P(X_i = x, Q) \quad = \quad & \sum_{n \in N(Q) \wedge v(n) = X_i} e^x(n) + \\ & p_{ix} \cdot \left(\sum_{n \in Del^0(X_i)} e^0(n) + \sum_{n \in Del^1(X_i)} e^1(n) \right) \end{aligned}$$

where p_{ix} is p_i if $x = 1$ and $(1 - p_i)$ if $x = 0$.

16.2.2 EDGE's Algorithm

Algorithm 16.1 shows the main procedure of EDGE. It takes as input a theory $\mathcal{K}$ a set of positive examples P_E, a set of negative examples N_E and two thresholds ϵ and δ. EDGE consists of an EM cycle, in which procedures EXPECTATION and MAXIMIZATION are repeatedly called until the log-likelihood of the examples converges to a local maximum. The former procedure returns the log-likelihood LL, the latter maximize it. The log-likelihood is obtained by Equation:

$$LL = \sum_{Q} \log P(Q)$$

The EM cycle stops when the difference between the LL of the current iteration and the one of the previous iteration drops below a threshold ϵ or when this difference is below a fraction δ of the previous log-likelihood.

Algorithm 16.1 Procedure EDGE

```
1: function EDGE(K, P_E, N_E, ε, δ)
2:     Build BDDs                          ▷ uses BUNDLE to build all the BDDs
3:     LL = −∞
4:     repeat
5:         LL_0 = LL
6:         LL = EXPECTATION(BDDs)
7:         MAXIMIZATION
8:     until LL − LL_0 < ε ∨ LL − LL_0 < −LL_0 · δ
9:     return LL, p_i for all i
10: end function
```

Algorithm 16.2 shows the procedure EXPECTATION. It takes as input a list of BDDs, one for each example (query), and computes the expectations for each one, i.e. $P(X_i = x, Q)$ for all variables X_i in the BDD. In this procedure we use $\eta^x(i)$ to indicate $P(X_i = x, Q)$. EXPECTATION first calls GETFORWARD and GETBACKWARD. They respectively compute the forward and the backward probability of nodes and $\eta^x(i)$ for non-deleted paths only. GETBACKWARD returns the probability of the query $P(Q)$, called $Prob$ in the procedure. Then, to take into account deleted paths, EXPECTATION updates $\eta^x(i)$.

Algorithm 16.2 Procedure EXPECTATION

```
function EXPECTATION(BDDs)
    LL = 0
    for all i ∈ Axioms do
        E[c_i0] = 0
        E[c_i1] = 0
    end for
    for all BDD ∈ BDDs do
        for all i ∈ Axioms do
            η^0(i) = 0
            η^1(i) = 0
        end for
        for all variables X do
            ς(X) = 0
        end for
        GETFORWARD(root(BDD))
        Prob=GETBACKWARD(root(BDD))
        T = 0
        for l = 1 to levels(BDD) do
            Let X_i be the variable associated with level l
            T = T + ς(X_i)
            η^0(i) = η^0(i) + T · (1 − p_i)
            η^1(i) = η^1(i) + T · p_i
        end for
        for all i ∈ Axioms do
            E[c_i0] = E[c_i0] + η^0(i)/Prob
            E[c_i1] = E[c_i1] + η^1(i)/Prob
        end for
        LL = LL + log(Prob)
    end for
    return LL
end function
```

Procedure MAXIMIZATION, shown in Algorithm 16.3, computes the parameter values for the next EM iteration.

Algorithm 16.3 Procedure MAXIMIZATION

```
function MAXIMIZATION
    for all i ∈ Axioms do
        p_i = E[c_i1] / (E[c_i0]+E[c_i1])
    end for
end function
```

As said before, procedure GETFORWARD (Algorithm 16.4), computes the value of the forward probabilities. Starting from the root level (where $F(root) = 1$)), it traverses the BDD one level at each iteration. For each level it considers each node n and computes its contribution to the forward probabilities of its children. Then the forward probabilities of its children, stored in table F, are updated.

Algorithm 16.4 Procedure GETFORWARD: computation of the forward probability

```
function GETFORWARD(root)
    F(root) = 1
    F(n) = 0 for all nodes
    for l = 1 to levels do                ▷ levels is the number of levels of the BDD rooted at root
        Nodes(l) = ∅
    end for
    Nodes(1) = {root}
    for l = 1 to levels do
        for all node ∈ Nodes(l) do
            Let X_i be v(node), the variable associated with node
            if child_0(node) is not terminal then
                F(child_0(node)) = F(child_0(node)) + F(node) · (1 − p_i)
                Add child_0(node) to Nodes(level(child_0(node)))    ▷ level(node) returns the level of node
            end if
            if child_1(node) is not terminal then
                F(child_1(node)) = F(child_1(node)) + F(node) · p_i
                Add child_1(node) to Nodes(level(child_1(node)))
            end if
        end for
    end for
end function
```

Algorithm 16.5 shows the procedure GETBACKWARD. This function computes the backward probability of nodes by traversing recursively the tree from the leaves to the root.

16.2.3 How to Use EDGE

EDGE is written in Java and has been integrated in DL-Learner 1.3 [170]. The latest stable version is 3.2 and it can be downloaded at `https://bitbucket.org/machinelearningunife/edge/downloads/`. It can be used as standalone desktop application. The user manual for EDGE can be found at `https://sites.google.com/a/unife.it/ml/edge/manual`.

If your application needs to perform parameter learning on DISPONTE KBs, EDGE can also be used as a library. If you are using Maven, all you have to do is to add the following lines in your `POM.xml`.

Algorithm 16.5 Procedure GetBackward: computation of the backward probability, updating of η and of ς

function GetBackward($node$)
  if $node$ is a terminal **then**
    return $value(node)$
  else
    Let X_i be $v(value)$
    $B(child_0(node)) =$ GetBackward($child_0(node)$)
    $B(child_1(node)) =$ GetBackward($child_1(node)$)
    $e^0(node) = F(node) \cdot B(child_0(node)) \cdot (1 - p_i)$
    $e^1(node) = F(node) \cdot B(child_1(node)) \cdot p_i$
    $\eta^0(i) = \eta_t^0(i) + e^0(node)$
    $\eta^1(i) = \eta_t^1(i) + e^1(node)$
    $VSucc = succ(v(node))$ ▷ $succ(X)$ returns the variable following X in the order
    $\varsigma(VSucc) = \varsigma(VSucc) + e^0(node) + e^1(node)$
    $\varsigma(v(child_0(node))) = \varsigma(v(child_0(node))) - e^0(node)$
    $\varsigma(v(child_1(node))) = \varsigma(v(child_1(node))) - e^1(node)$
    return $B(child_0(node)) \cdot (1 - p_i) + B(child_1(node)) \cdot p_i$
  end if
end function

```
<dependency>
    <groupId>unife</groupId>
    <artifactId>edge</artifactId>
    <version>3.2</version>
    <exclusions>
        <exclusion>
            <groupId>${project.groupId}</
               groupId>
            <artifactId>manpageGenerator-
               maven-plugin</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

16.3 Conclusion

EDGE is a parameter learning algorithm for PDLs that follow DISPONTE. In [14] the authors compared EDGE with Association Rules (ARs) and the experimental results showed that EDGE has better performances. In this chapter we investigated the details of EDGE in order to be able to understand the next chapter (Chapter 17), which presents a distributed version of EDGE, called EDGE$^{\text{MR}}$.

Chapter 17

Distributed Parameter Learning for Probabilistic Description Logics

In this chapter we illustrate an algorithm for distributed parameter learning on PDLs that follow DISPONTE named EDGE^{MR}. After an introduction of the motivations in Section 17.1 that led to the development of EDGE^{MR}, this algorithm is explained in Section 17.2. Section 17.3 shows the results of the experiments for evaluating EDGE^{MR}. Finally, Section 17.4 draws conclusions.

17.1 Introduction

In Chapter 9 we illustrated DISPONTE, which adapts the distribution semantics for probabilistic logic programming to DLs and allows to integrate probabilistic information in DLs.

In Chapter 16 we illustrated EDGE [14], an algorithm for learning the parameters of PDLs following DISPONTE. EDGE was tested on various datasets and was able to find good solutions. However, like EMBLEM, the execution of this algorithm is rather expensive from a computational point of view. In order to reduce EDGE running time, we developed EDGE^{MR}, which represents a MapReduce implementation of EDGE.

Various MapReduce frameworks are available, such as Hadoop. However, as for EMBLEM and SEMPRE, we chose not to use any framework and to implement a MapReduce approach for EDGE^{MR} based on the Message Passing Interface (MPI).

17.2 Distributed Parameter Learning: EDGE^{MR}

Like most MapReduce frameworks, EDGE^{MR} architecture follows a master-slave model. The communication between the master and the slaves adopts the Message Passing Interface (MPI), in particular we used the OpenMPI[1] library which provides a Java interface to the native library. The processes of EDGE^{MR} are not purely functional, as required by standard MapReduce frameworks such as Hadoop, because they have to retain in main memory the BDDs during the whole execution. This forced us to develop a parallelization strategy exploiting MPI.

[1] http://www.open-mpi.org/

EDGE^{MR} can be split into three phases: *Initialization*, *Query resolution* and *Expectation-Maximization*. All these operations are executed in parallel and synchronized by the master.

Initialization During this phase data is replicated and a process is created on each machine. Then each process parses its copy of the probabilistic knowledge base and stores it in main memory. The master, in addition, parses the files containing the positive and negative examples (the queries).

Query resolution The master divides the set of queries into subsets and distributes them among the workers. Each worker generates its private subset of BDDs and keeps them in memory for the whole execution. Two different scheduling techniques can be applied for this operation. See Subsection 17.2.2 for details.

Expectation-Maximization After all the nodes have built the BDDs for their queries, EDGE^{MR} starts the Expectation-Maximization cycle. During the Expectation step all the workers traverse their BDDs and calculate their local *eta* array. Then the master gathers all the *eta*'s from the workers and aggregates them by summing the arrays component-wise. Then it calls the Maximization procedure in which it updates the parameters and sends them to the slaves. The cycle is repeated until one of the stopping criteria is satisfied.

17.2.1 MapReduce View

Since EDGE^{MR} is based on MapReduce, it can be split into three phases: *Initialization*, *Map* and *Reduce*.

Initialization As described above.

Map This phase can be seen as a function that returns a set of *(key, value)* pairs, where *key* is an example identifier and *value* is the array *eta*.

- *Query resolution*: each worker resolves its chunks of queries and builds its private set of BDDs. Two different scheduling techniques can be applied for this operation. See Subsection 17.2.2 for details.
- *Expectation Step*: each worker calculates its local *eta*.

Reduce This phase is performed by the master (also referred to as the "reducer") and it can be seen as a function that returns pairs (i, p_i), where i is an axiom identifier and p_i is its probability.

- *Maximization Step*: the master gathers all the *eta* arrays from the workers, aggregates them by summing component wise, performs the Maximization step and sends the newly updated parameters to the slaves.

The Map and Reduce phases implement the Expectation and Maximization functions respectively, hence they are repeated until a local maximum is reached. It is important to notice that the Query Resolution step in the Map phase is executed only once because the workers keep in memory the generated BDDs for the whole execution of the EM cycle; what changes among iterations are the random variables' parameters.

17.2.2 Scheduling Techniques

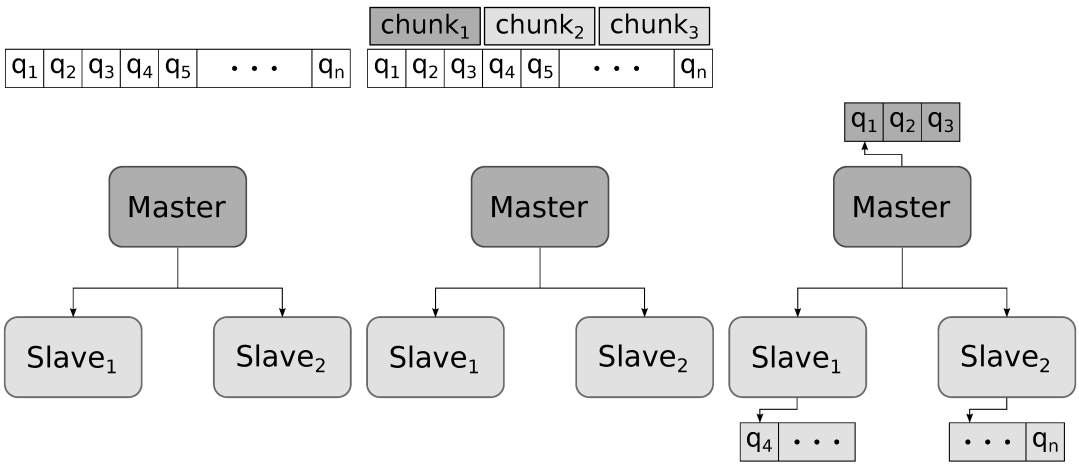
In a distributed context the scheduling strategy influences significantly the performances. We evaluated two scheduling strategies, *single-step scheduling* and *dynamic scheduling*, during the generation of the BDDs for the queries, while the initialization and the EM phases are independent of the chosen scheduling method.

Single-step Scheduling if N is the number of the slaves, the master divides the total number of queries into $N + 1$ chunks, i.e. the number of slaves plus the master. Then the master starts $N + 1$ threads, one building the BDD for its queries while the others sending the other chunks to the corresponding slaves. After the master has terminated dealing with its queries, it waits for the results from the slaves. When the slowest slave returns its results to the master, EDGE^{MR} proceeds to the EM cycle. Figure 17.1a shows an example of single-step scheduling with two slaves.

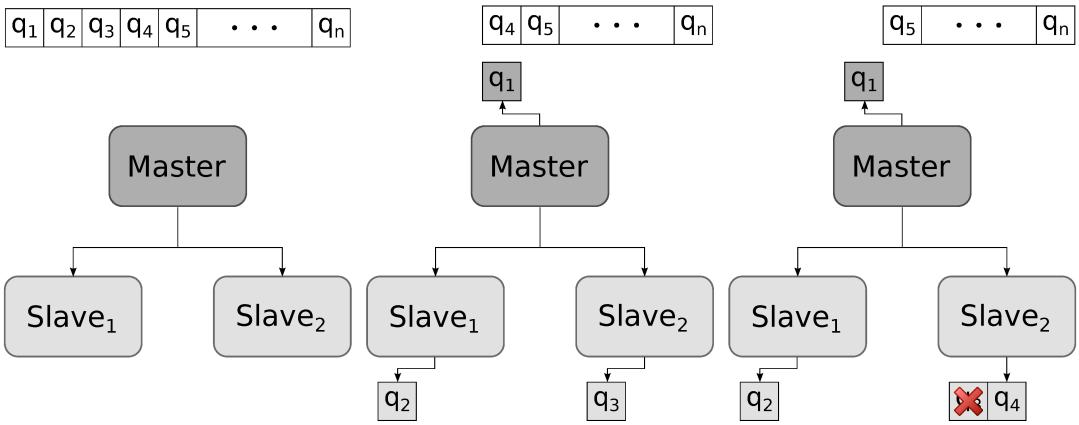
Dynamic Scheduling is more flexible and adaptive than single-step scheduling. Handling each query chunk may require a different amount of time. Therefore, with single-step scheduling, it could happen that a slave takes much more time than another one to deal with its chunk of queries. This may cause the master and some slaves to wait. Dynamic scheduling mitigates this issue. The user can establish a chunk dimension, i.e. the number of examples in each chunk. At first, each machine is assigned a chunk of queries in order. When it finishes handling the chunk, it takes the following chunk. So if the master ends handling its chunk, it just picks the next one, while if a slave ends handling its chunk, it asks the master for another one. During this phase the master runs a listener thread that waits for slaves' requests of new chunks. For each request, the listener starts a new thread that sends a chunk to the requesting slave (to improve the performances this is done through a thread pool). When all the BDDs for the queries are built, EDGE^{MR} starts the EM cycle. An example of dynamic scheduling with two slaves and a chunk dimension of one example is displayed in Figure 17.1b.

17.2.3 EDGE^{MR}'s Algorithm

EDGE^{MR}'s main procedure is shown in Algorithm 17.1. First each process reads the given input, every process has a copy of the knowledge base. Then the master, using the scheduling chosen by the user, sends the examples to the slaves. The master keeps a part of the examples and builds its BDDs (lines 14-27). Here, in particular, if dynamic scheduling is chosen, the master initializes a thread listener (line 17) which sends a chunk of examples to the slaves at every request it receives. The slaves, according to the scheduling technique, receive the examples and build the corresponding BDDs (lines 40-49). All the BDDs are built by using BUNDLE, where NL and TL represent the maximum number of explanations and the reasoning time limit. Once all the BDDs are built, we don't need to perform reasoning anymore, but we have to perform the EM algorithm. The master sends the probability values p_i to the slaves (line 31) which receive and store them (line 51). Now, the EXPECTATION procedure (Algorithm 16.2) can be executed by all the workers (lines 32 and 52). Finally, all workers enter in the



(a) Single-step scheduling



(b) Dynamic scheduling

Figure 17.1: Scheduling techniques of EDGE^{MR}.

maximization phase where the master collects all the values, executes the MAXIMIZATION procedure (Algorithm 16.3) and checks whether a new round of EM must be performed according to the thresholds ϵ and δ (line 33-37), while the slaves only wait for a signal from master which indicates whether to execute either EXPECTATION or stop.

Algorithm 17.1 Function EDGE$^{\mathrm{MR}}$

```
1: function EDGE^MR(K, E+, E−, ϵ, δ, NL, TL, S)
2:     Input: a knowledge base K
3:     Input: a set of positive examples E+
4:     Input: a set of negative examples E−
5:     Input: a threshold ϵ for the difference between LLs
6:     Input: a threshold δ for the fraction of the difference between LLs
7:     Input: the maximum number of explanations to find for each example NL
8:     Input: the time limit for the inference process for each example TL
9:     Input: the scheduling method S
10:    Output: the final LL
11:    Output: probabilities p_i of the probabilistic axioms
12:    Read knowledge base K
13:    if MASTER then
14:        Identify examples E
15:        if S == dynamic then                                   ▷ dynamic scheduling
16:            Send an example e_j to each slave
17:            Start thread listener          ▷ Thread for answering query requests from slaves
18:            c = m − 1                                  ▷ c counts the computed examples
19:            while c < |E| do
20:                c = c + 1
21:                Build BDD_c for example e_c                    ▷ performed by BUNDLE
22:            end while
23:        else                                                   ▷ single-step scheduling
24:            Split examples E into n subsets E_1, ..., E_n
25:            Send E_m to each worker m, 2 ≤ m ≤ n
26:            Build BDDs_1 for examples E_1
27:        end if
28:        LL = −∞
29:        repeat
30:            LL_0 = LL
31:            Send the parameters p_i to each worker m, 2 ≤ m ≤ n
32:            LL = Expectation(BDDs_1)
33:            Collect LL_m and the expectations from each worker m, 2 ≤ m ≤ n
34:            Update LL and the expectations
35:            Maximization
36:        until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
37:        Send STOP signal to all slaves
38:        return LL, p_i for all i
39:    else                                                       ▷ the j-th slave
40:        if S == dynamic then                                   ▷ dynamic scheduling
41:            while c < |E| do
42:                Receive e_j from master
43:                Build BDD_j for example e_j
44:                Request another example to the master
45:            end while
46:        else                                                   ▷ single-step scheduling
47:            Receive E_j from master
48:            Build BDDs_j for examples E_j
49:        end if
50:        repeat
51:            Receive the parameters p_i from master
52:            LL_j = Expectation(BDDs_j)
53:            Send LL_j and the expectations to master
54:        until Receive STOP signal from master
55:    end if
56: end function
```

17.3 Experiments

In order to evaluate the performances of EDGE^{MR}, four datasets were selected:

- **Mutagenesis**[2] [225], contains information about a number of aromatic and heteroaromatic nitro drugs, including their chemical structures in terms of atoms, bonds and a number of molecular substructures.
- **Carcinogenesis**[3] [227], which describes the carcinogenicity of more than 300 chemical compounds.
- an extract of **DBPedia**[4] [238], a knowledge base obtained by extracting the structured data from Wikipedia.
- `education.data.gov.uk`[5], which contains information about school institutions in the United Kingdom.

The last three datasets are the same as in [16]. All experiments have been performed on a cluster of 64-bit Linux machines with 2 GB (max) memory allotted to Java per node. Each node of this cluster has 8-cores Intel Haswell 2.40 GHz CPUs.

For the generation of positive and negative examples, we randomly chose a set of individuals from the dataset. Then, for each extracted individual a, we sampled three named classes: A and B were selected among the named classes to which a explicitly belongs, while C was taken from the named classes to which a does not explicitly belong but that exhibits at least one explanation for the query $a : C$. The axiom $a : A$ was added to the KB, while $a : B$ was considered as a positive example and $a : C$ as a negative example. Then both the positive and the negative examples were split in five equally sized subsets and we performed five-fold cross-validation for each dataset and for each number of workers. Information about the datasets and training examples is shown in Table 17.1. We performed the experiments with 1, 3, 5, 9 and

Table 17.1: Characteristics of the datasets used for evaluation.

Dataset	**# of all axioms**	**# of probabilistic axioms**	**# of pos. examples**	**# of neg. examples**	**Fold size (MiB)**
Carcinogenesis	74409	186	103	154	18.64
DBpedia	5380	1379	181	174	0.98
`education.data.gov.uk`	5467	217	961	966	1.03
Mutagenesis	48354	92	500	500	6.01

17 nodes, where the execution with 1 node corresponds to the execution of EDGE. Furthermore, we used both single-step and dynamic scheduling in order to evaluate the two scheduling approaches. It is important to point out that the quality of the learning is independent of the type of scheduling and of the number of nodes, i.e. the

[2] http://www.doc.ic.ac.uk/ shm/mutagenesis.html
[3] http://dl-learner.org/wiki/Carcinogenesis
[4] http://dbpedia.org/
[5] http://education.data.gov.uk

parameters found with 1 node are the same as those found with n nodes. Table 17.2 shows the running time in seconds for parameter learning on the four datasets with the different configurations. Figure 17.2 shows the speedup obtained as a function of the

Table 17.2: Comparison between EDGE and EDGE$^{\text{MR}}$ in terms of running time (in seconds) for parameter learning.

Dataset	**EDGE**	**EDGE$^{\text{MR}}$**							
		Dynamic				**Single-step**			
		3	5	9	17	3	5	9	17
Carcinogenesis	847	442	241	147	94	384	268	179	118
DBpedia	1552	1260	634	365	215	1156	724	453	373
education.data.gov.uk	6924	3878	2157	1086	623	3612	2290	1332	749
Mutagenesis	1439	636	400	223	130	578	359	230	125

number of machines (nodes). The speedup is the ratio of the running time of 1 worker to the running time of n workers. We can note that the speedup is significant even if it is sublinear, showing that a certain amount of overhead (the resources, and therefore the time, spent for the MPI communications) is present. The dynamic scheduling technique has generally better performance than single-step scheduling.

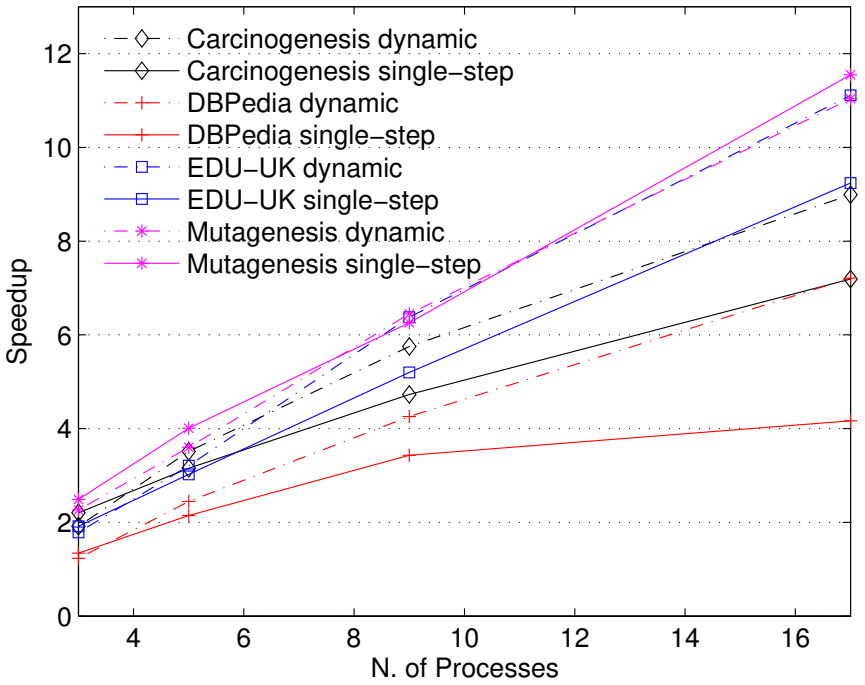
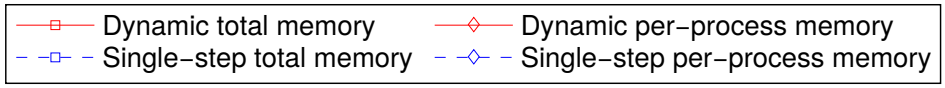


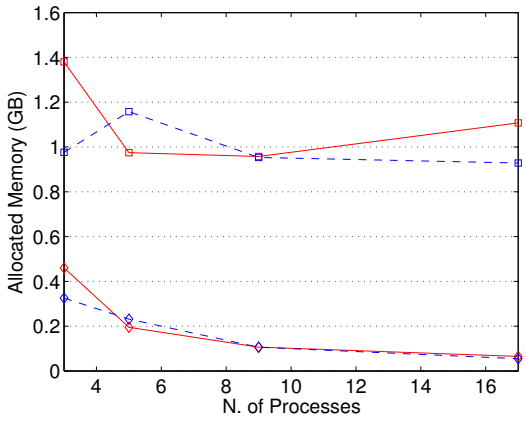
Figure 17.2: Speedup of EDGE$^{\text{MR}}$ relative to EDGE with single-step and dynamic schedulings.

We also tested memory consumption of EDGE$^{\text{MR}}$ on these datasets. The results, shown in Fig. 17.3, show that the allocated memory per node is almost always inversely

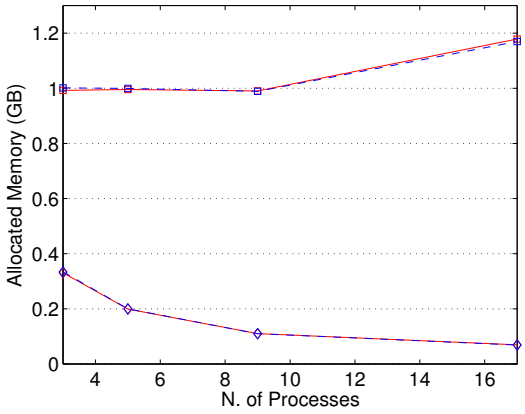
proportional to the number of nodes. There is no difference between Single-step and Dynamic scheduling in terms of used memory.



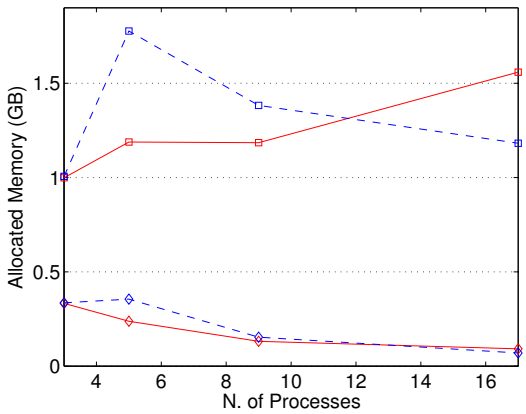
(a) Legend for memory consumption graphs



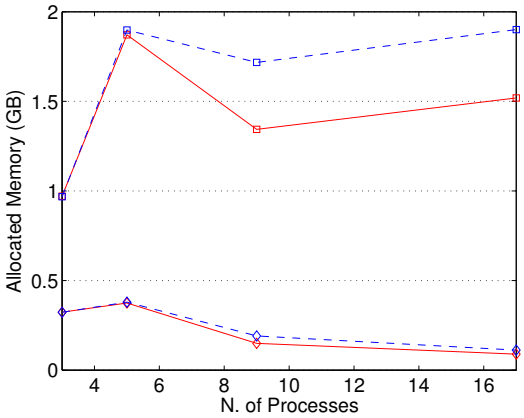
(b) Carcinogenesis



(c) DBPedia



(d) EDU-UK



(e) Mutagenesis

Figure 17.3: Memory consumption of EDGE^{MR} for different datasets.

17.4 Conclusions

EDGE is an algorithm for learning the parameters of probabilistic knowledge bases under the DISPONTE semantics. In this chapter we presented EDGE^{MR}, which is a distributed version of EDGE based on the MapReduce approach.

We performed experiments over four datasets with an increasing number of nodes. The results show that parallelization significantly reduces the execution time, even if

in a sublinear trend. The sublinearity is caused by the overhead and because some threads are devoted to communication tasks.

In the next chapter we introduce a structure learning algorithm for PDLs called LEAP.

Chapter 18

Structure Learning in Probabilistic Description Logics

In this chapter we present the LEAP system, which performs structure learning on PDLs that follow DISPONTE. The chapter is organized as follows. Section 18.1 introduces the chapter. Section 18.2 illustrates the learning problem. Section 18.3 provides an overview of the theory of the refinement operators for DLs. Section 18.4 quickly describes CELOE [239]. Section 18.5 depicts in a few words DL-Learner [170], an open source project which contains and implements CELOE, the refinement operators and several other components. Section 18.6 is the central section of this chapter where we illustrate LEAP. In Section 18.7 and Section 18.8 we discuss related work and the experimental results respectively. Finally, section 18.9 draws conclusions.

18.1 Introduction

In Chapters 16 and 17 we investigated how to perform parameter learning on PDLs in a sequential and distributed way. As mentioned before, the other learning task of SRL is structure learning. To solve this problem we developed LEAP.

The Semantic Web is becoming more and more widespread, but despite its diffusion, there are still several knowledge bases that does not have refined structures. A knowledge base with a refined structure, and instance data coherent with it, allows more powerful reasoning and better consistency checking. Therefore strategies for automated structure building of ontologies are beginning to come out.

In order to learning the structure, i.e. new probabilistic axioms, LEAP combines two algorithms, CELOE [239] and EDGE. It first finds good candidate axioms (subsumption axioms) by means of CELOE, then it performs a greedy search in the space of theories by exploiting EDGE for learning the parameter of the probabilistic KB.

In Chapter 16 we already described EDGE, in this chapter we give a brief overview of CELOE and the theory that underlies it. CELOE is a learning algorithm whose goal is to provide a semi-automatic method to **learn class expressions** of target concepts and hence improve the structure of the background knowledge base.

LEAP is written in Java and is now part of DL-Learner since version 1.3.

18.2 The Learning Problem

In this section we briefly present the concept learning problem in description logics. The definition here provided is a particular case of the definition of the ILP problem given in Definition 14.1, when the target predicate is unary.

Learning from entailment problem in description Logics can be defined as follows.

Definition 18.1 Concept Learning from Entailment Problem in Description Logics
Given:

- a concept name Target;
- a background knowledge base $\mathcal{K}$ not containing Target;
- a space of possible concepts $\mathcal{C}$;
- a set of positive examples $\mathcal{E}^+$ with elements of the form $a : \mathsf{Target}$ $(a \in \mathbf{I})$;
- a set of negative examples $\mathcal{E}^-$ with elements of the form $a : \mathsf{Target}$ $(a \in \mathbf{I})$.

Find a concept $C \in \mathcal{C}$ such that:

- $\mathsf{Target} \equiv C$;
- Target does not occur in C (acyclic definition);
- $\forall e^+ \in \mathcal{E}^+, \mathcal{K} \cup \{\mathsf{Target} \equiv C\} \models e^+$;
- $\forall e^- \in \mathcal{E}^-, \mathcal{K} \cup \{\mathsf{Target} \equiv C\} \not\models e^-$.

□

Let $\mathcal{K}' = \mathcal{K} \cup \{\mathsf{Target} \equiv C\}$ we say that a concept C *covers* an example $e \in \mathcal{E}^+ \cup \mathcal{E}^-$ if $\mathcal{K}' \models e$. Here, if both sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of individuals are given the problem takes the name of *Positive and Negative Examples Learning Problem*, while if only the set $\mathcal{E}^+$ is available it is called *Positive Examples Learning Problem*. Thus, the goal of learning is to discover a correct concept C with respect to the individuals.

18.3 Refinement Operators in Description Logics

Finding a correct concept can be seen as a search process in the space of possible concepts. In Inductive Logic Programming a well-known approach is to impose an ordering on the search space of hypotheses and use the **refinement operators** to traverse it. *Downward (upward) refinement operators* produce *specialization* (*generalization*) of hypotheses (see Section 14.2).

The subsumption relation $\sqsubseteq$ is a quasi-ordering, hence it can be used for ordering the search space and the refinements operators can be used to search the space. We can define the refinement operator for description logics as follows.

Definition 18.2 Refinement Operator in Description Logics
Let $\mathcal{L}_r$ be a description logic. A refinement operator in the quasi-ordered space $(\mathcal{L}_r, \sqsubseteq)$ is called an $\mathcal{L}_r$ *refinement operator*. □

It is of fundamental importance to say that the description logic of the background knowledge base can be more expressive than the description language of the learned concept C. For example we can learn $\mathcal{ALCQ}$ concepts, but the language of the knowledge base can be $\mathcal{SHOIN}(\mathbf{D})$ or $\mathcal{SROIQ}(\mathbf{D})$.

In [240] Jens Lehmann et al. provided a theoretical investigation of the properties of $\mathcal{ALCQ}$ refinement operators. These properties are then exploited for the definition and the construction of suitable refinement operators. The theory exposed in [240] is based on earlier work [241, 242].

Below we illustrate a quick overview of the theoretical analysis provided in [240] for $\mathcal{ALCQ}$ refinement operators. **The definitions and the theorems exposed in the rest of this section are from [240, 241, 242]**.

Definition 18.3 Refinement Chain
A refinement chain of an $\mathcal{L}_r$ refinement operator ρ of length n from a concept C to a concept D is a finite concatenation $C_0, C_1, \ldots, C_n$ of concepts, where $C = C_0, C_1 \in \rho(C_0), C_2 \in \rho(C_1), \ldots, C_n \in \rho(C_{n-1}), D = C_n$. We say that D can be reached from C. This refinement chain *goes through* E iff there is a C_i with $1 \leq i \leq n$ such that $E = C_i$. $\rho^*(C)$ identifies the set of all the concepts, which can be reached from C by ρ. $\rho^m(C)$ denotes the set of all the concepts reachable from C by a refinement chain of ρ of length m. □

Definition 18.4 Downward and Upward Cover
Let C, D and E be concept.

- A concept C is a *downward cover* of a concept D iff $C \sqsubset D$ and there does not exist a concept E such that $C \sqsubset E \sqsubset D$.
- A concept C is an *upward cover* of a concept D iff $D \sqsubset C$ and there does not exist a concept E such that $D \sqsubset E \sqsubset C$.

□

Definition 18.5 equivalence, (syntactic equality, weak equality)
Let C, D be two concepts.

Equivalence C and D are *equivalent* $C \equiv D$ if they identify the same subset of individuals.

Equality C and D are *equal* if they are syntactically equal.

Weak equality C and D are *weakly equal*, denoted by $C \simeq D$ iff they are equal up to permutation of arguments of conjunction and disjunction. For example $\mathsf{Male} \sqcap \exists \mathsf{hasChild}.\top \simeq \exists \mathsf{hasChild}.\top \sqcap \mathsf{Male}$

□

A refinement operator can have various properties which could influence the performance and the quality of the learning process.

Definition 18.6 Properties of DL refinement operators
An $\mathcal{L}_r$ refinement operator ρ is called

- (*locally*) *finite* iff $\rho(C)$ is finite for all concept C.
- *redundant* iff there exists a refinement chain from a concept C to a concept D, which does not go through some concept E and a refinement chain from C to a concept D', which goes through E and $D' \simeq D$.
- *proper* iff for all concepts C and D, $D \in \rho(C) \implies C \not\equiv D$.

An $\mathcal{L}_r$ downward refinement operator ρ is called

- *complete* iff for all concepts C, D with $C \sqsubset D$ we can reach a concept E with $E \equiv C$ from D by ρ.
- *weakly complete* iff for all concepts $C \sqsubset \top$ we can reach a concept E with $E \equiv C$ from $\top$ by ρ.
- *minimal* iff for all C, $\rho(C)$ contains only downward covers and all its elements are incomparable with respect to $\sqsubseteq$.

An $\mathcal{L}_r$ upward refinement operator ρ is called

- *complete* iff for all concepts C, D with $D \sqsubset C$ we can reach a concept E with $E \equiv C$ from D by ρ.
- *weakly complete* iff for all concepts $\bot \sqsubset C$ we can reach a concept E with $E \equiv C$ from $\bot$ by ρ.
- *minimal* iff for all C, $\rho(C)$ contains only upward covers and all its elements are incomparable with respect to $\sqsubseteq$.

An $\mathcal{L}_r$ refinement operator ρ is called

- *ideal* iff it is finite, complete and proper.

□

The following theorem if essential, in order to design a suitable refinement operator.

Theorem 18.1
Considering the properties of completeness, weak completeness, properness, finiteness, and non-redundancy, the following are maximal sets of properties (in the sense that no other of the mentioned properties can be added) of $\mathcal{L}_r$ refinement operators ($\mathcal{L}_r \in \{\mathcal{ALC}, \mathcal{ALCQ}, \mathcal{SHOIN}, \mathcal{SROIQ}\}$):

1. *{weakly complete, complete, finite}*

2. *{weakly complete, complete, proper}*
3. *{weakly complete, non-redundant, finite}*
4. *{weakly complete, non-redundant, proper}*
5. *{non-redundant, finite, proper}*

The authors of [240] proposed a refinement operator that respects the properties listed in point 2. The refinement operator is redundant and infinite. Redundancy elimination and infinity handling are tasks that are left to the learning algorithm. To improve the performance of the refinement operator ρ, a learning algorithm can implement a method that transforms the subsumption graph[1] into two trees: one tree for downward refinement of atomic concepts and one tree for upward refinement of atomic concepts[2].

18.4 CELOE

CELOE [239] stands for "Class Expression Learning for Ontology Engineering". It is a learning algorithm implemented within the open-source framework DL-Learner[3].

Let us consider a knowledge base $\mathcal{K}$ and a class Target whose formal description we want to learn. Target has (inferred or asserted) instances in the knowledge base $\mathcal{K}$. If Target is already described by a class expression C through axioms such as $\mathsf{Target} \sqsubseteq C$ or $\mathsf{Target} \equiv C$, it is possible to learn a description for Target by refining C, or by relearning from scratch.

Definition 18.7 Class Learning Problem
Let an existing named class Target be in a knowledge base $\mathcal{K}$. Let $R_{\mathcal{K}}(C)$ be a *retrieval* reasoner operation that returns the set of all instances of C. The **class learning problem** is to find an expression C such that $R_{\mathcal{K}}(\mathsf{Target}) = R_{\mathcal{K}}(C)$. □

CELOE is a semi-automatic approach, that finds a set of n class expressions C_i $(1 \leq i \leq n)$ sorted according to a heuristic. Such expressions are candidates for adding axioms of the form $\mathsf{Target} \equiv C_i$ or $\mathsf{Target} \sqsubseteq C_i$. It is a decision of the knowledge base engineer to add or not these axioms to the background knowledge base. CELOE can take as input a target class, a set of positive and negative examples or a set of only positive examples. When a target class is defined, CELOE can be seen as a learning algorithm that resolves the class learning problem as determined in Definition 18.7. On the other hand, if a set of positive and negative examples or a set of only positive examples is given, CELOE can be seen as a learning algorithm that resolves the learning problem described in Definition 18.1.

A learning algorithm can be built as a combination of a *refinement operator* and a search algorithm. The former determines how the search tree can be built, the latter controls how the tree is traversed.

[1] a quasi-ordered space identifies a directed graph

[2] Indeed, inside DL-Learner, there is the method `ClassHierarchy#thinOutSubsumptionHierarchy()` that does this job

[3] http://dl-learner.org/Projects/DLLearner

CELOE is a top-down algorithm that uses the $\mathcal{ALCQ}$ refinement operator defined in [240] as illustrated in Figure 18.1. Each generated class expression is evaluated using

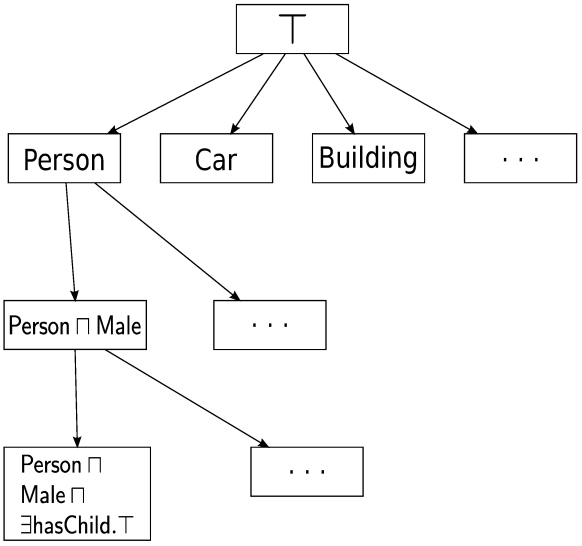


Figure 18.1: Illustration of a search tree in a top down refinement approach.

a heuristic, then the resulting values are used to guide the search in a learning process. CELOE can use one of five different heuristics: *Predictive Accuracy*, *F-Measure*, *A-Measure*, *Generalized F-Measure* and *Jaccard Distance*. All these heuristics need a set of examples in order to be computed. If the knowledge engineer gave as input a set of positive and negative examples or only a set of positive examples, there are no problems, but if she gave as input a target class to describe, it is not possible to compute the heuristic. To overcome this problem we can consider as positive examples the existing instances (inferred or asserted) of the target class and the remaining instances as negative examples. This is illustrated in Figure 18.2, $\mathcal{K}$ is the knowledge base, A the target class that we want to describe and C the found class expression. $R_{\mathcal{K}}(A) \setminus R_{\mathcal{K}}(C)$

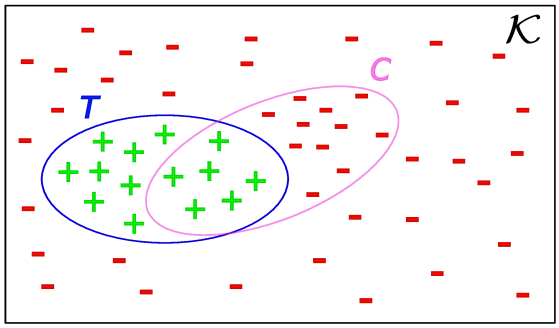


Figure 18.2: Positive and negative examples in a class learning problem. $\mathcal{K}$ is the knowledge base, A the class to describe and C the class expression to be tested.

are the false negatives, $R_{\mathcal{K}}(A) \cap R_{\mathcal{K}}(C)$ are the true positives, $R_{\mathcal{K}}(C) \setminus R_{\mathcal{K}}(A)$ are the

false positive and the remaining instances ($R_{\mathcal{K}}(\top) \setminus (R_{\mathcal{K}}(C) \cup R_{\mathcal{K}}(A))$) are the true negatives.

Performing instance retrieval can be very expensive for large ontologies. In order to make CELOE scalable, three performance optimizations are provided:

Reduction of instance checks: it consists of the reduction of the number of objects we are looking at by using background knowledge. Let consider the case we want to learn an equivalence axiom for class A which has a super class A'. It is useful to start the top-down search with A' instead of $\top$. In this way the number of negative examples is lower.

Approximate and closed world reasoning: it consists of using a reasoner designed for performing a high number of instance checks. The authors of CELOE created an approximate incomplete reasoning procedure for fast instance checks (FIC) which partially follows the closed world assumption.

Stochastic coverage computation: randomly drawn objects are tested until a fixed width of the interval of confidence is reached. The confidence interval is computed by using the improved Wald method defined in [243]. See [239] and [243] for further details.

18.5 DL-Learner

From the DL-Learner [170] manual[4]:

> *DL-Learner is a machine learning framework for OWL and description logics. It includes several learning algorithms and is easy to extend. DL-Learner widens the scope of Inductive Logic Programming to description logics and the Semantic Web.*

DL-Learner is written in Java, therefore it can be used on almost all operative systems.

In order to be flexible and easily extensible, DL-Learner uses a modular architecture shown in Figure 18.3. There are four types of components: knowledge sources, reasoners, learning problems and learning algorithms. For each type there are several implemented components. Each component can have various configuration options that are used to change the parameters/settings of a component.

- **Knowledge Sources** integrate background knowledge. DL-Learner supports all the OWL formats: RDF/XML, OWL/XML, Manchester OWL Syntax, or Turtle. Furthermore it allows to use SPARQL endpoints as background knowledge.

- **Reasoner Components** provide connections to existing or own reasoners. Several reasoner are implemented. OWL API and DIG interfaces can be used in conjunction with the major reasoners such as Pellet, FaCT++, etcetera. In addiction, DL-Learner offers its own reasoner based on Pellet: Fast Instance Checker. This reasoner partially follows the closed world assumption (CWA) and hence it is not correct with respect to the OWL semantics.

[4] `https://dl-learner.org/Resources/Documents/manual.pdf`

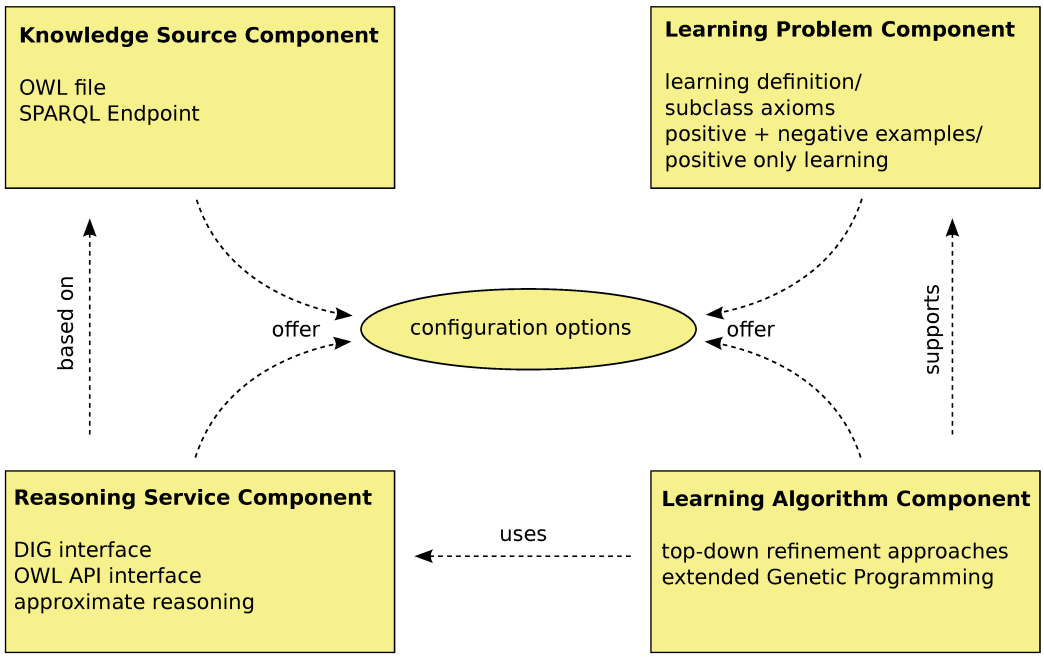


Figure 18.3: The architecture of DL-Learner (redrawing from [244]).

- **Learning Problems** specify the problem type, which is to be solved by an algorithm. Three type of learning problem are implemented: learning from positive and negative examples, positive-only learning and class learning.
- **Learning Algorithms** provide methods to solve one or more specified learning problem types. The implemented algorithms go from very simple algorithms (e.g. Brute Force) to sophisticated ones. CELOE belongs to this component category.

DL-Learner has a command line interface, a graphical user interface and a web service. For our purpose we will use only the command line.

18.6 Structure Learning: LEAP

LEAP learns both the structure and the parameters of DISPONTE knowledge bases.

In order to learn a KB, LEAP first finds good candidate axioms then it performs a greedy search in the space of theories.

18.6.1 Architecture

LEAP tries to combine CELOE with EDGE. The architecture is illustrated in Figure 18.4. The core components of LEAP are CELOE, EDGE and BUNDLE. The components in blue are the ones that natively take into account the probabilistic values. The components in yellow, instead, do not handle probabilistic axioms.

After CELOE has generated a set of class expressions, for each element, CELOE sends to EDGE the axiom to be added to the ontology. After each addition, EDGE

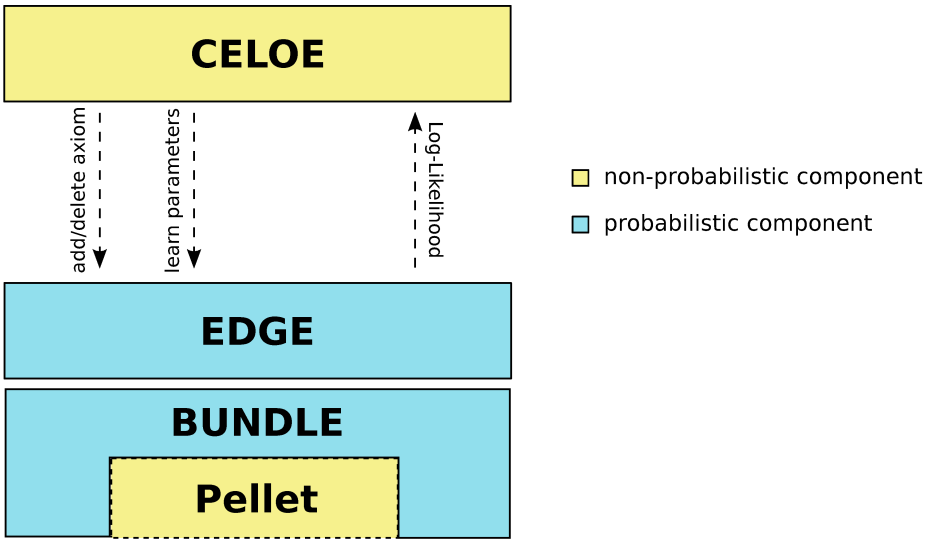


Figure 18.4: LEAP Architecture

is run on the modified ontology and the log-likelihood is computed. If the resulting log-likelihood is better then the currently best, the axiom is kept.

The next subsection will try to clarify the interaction between CELOE and EDGE.

18.6.2 Interfacing CELOE and EDGE

As already mentioned, CELOE takes as input a set of individuals or a target class and generates a set of class expressions, instead EDGE takes as input a set of examples (assertional axioms) and computes the parameters of the target probabilistic axioms.

Let us suppose that the knowledge engineer has given as input to CELOE a set of positive individuals P_I and a set of negative individuals N_I and CELOE has generated a set of class expressions $ClassExpressions$. In this case, which are the examples for EDGE and which are the probabilistic axioms whose parameters we want to compute? The set of positive and negative examples is composed of assertional axioms of the form $ind : dummyClass$, where $ind \in P_I \cup N_I$ and $dummyClass$ is a class created ad-hoc for EDGE. In order to generate a set of probabilistic axioms that must be evaluated by the parameter learner, we create a probabilistic subsumption axiom of the form $CE \sqsubseteq dummyClass$, for each $CE \in ClassExpressions$.

18.6.3 LEAP

LEAP's main procedure is shown in Algorithm 18.1. It takes as input the KB $\mathcal{K}$ and the type of learning problem LP_{type}; the maximum number of class expressions NC and the time limit TLC for CELOE; the values of ϵ and δ, the maximum number of explanations NE and the time limit TLE for the computation of the BDDs for each example for EDGE[5].

[5]Default values are: $NC = 10$, $TLC = 10$ seconds and $NE = TLE = \infty$

First of all, a set of class expressions are generated by using CELOE (line 11), then the positive (P_I) and negative (N_I) set of individuals is extracted according to the following rules (see section 4.3 of the DL-Learner manual).

- If a set of positive and negative individuals has been given as input to CELOE, then no extraction is necessary. LP_{type} = *Positive and Negative Examples Learning Problem.*
- if a set of positive only individuals has been given, then the set of positive examples is already defined, the set of negative examples is composed of a subset of all the individuals of the knowledge base except the positive ones. LP_{type} = *Positive Examples Learning Problem.*
- if a target class to be described is given, then we consider the existing instances (inferred or asserted) of the target class as positive individuals and the remaining instances as negative individuals. LP_{type} = *Class Learning Problem* (cf. Definition 18.7)

After the extraction, the assertional axioms, which represent the examples (queries) for EDGE, are created (lines 13-18). The $dummyClass$ is a class created *ad-hoc*[6] in order to run EDGE. The initial ontology is backed up (line 19).

Then LEAP performs a greedy search in the space of theories, described in lines 20-29. One subsumption axiom of the form $p :: CE \sqsubseteq dummyClass$ is added at a time to the ontology $\mathcal{K}$, where p is a random probabilistic value. After each addition, EDGE is run on the extended theory $\mathcal{K}' = \mathcal{K} \cup \{CE \sqsubseteq dummyClass\}$ and the log-likelihood LL and the new parameters of the probabilistic axioms are computed (line 23. If LL is better than the current best LL_0, the axiom is kept in the knowledge base and the list $LearnedAxioms$ is updated, otherwise the added axiom is discarded.

Finally, if the knowledge engineer has defined a target class, the right-hand side of each learned subsumption axiom is replaced with the target class (cf. lines 30-34). The final theory, obtained from the union of the initial ontology and the probabilistic learned axioms, is returned to the user (line 35).

[6]The name of this class is learnedClass, see the source code for details.

Algorithm 18.1 LEAP Algorithm

procedure LEAP($\mathcal{K}, LP_{type}, NC, TLC, \epsilon, \delta, NE, TLE$)
  Input: a knowledge base $\mathcal{K}$
  Input: the type LP_{type} of learning problem
  Input: the maximum number of class expressions to find NC
  Input: the time limit for the inference for CELOE TLC
  Input: a threshold ϵ for the difference between LLs
  Input: a threshold δ for the fraction of the difference between LLs
  Input: the maximum number of explanations to find for each example NE
  Input: the time limit for the inference process for each example TLE
  Output: the learned knowledge base $\mathcal{K}$
  Generate up to NC $ClassExpressions$ ▷ generated by CELOE
  $(P_I, N_I) = \text{ExtractIndividuals}(LP_{type})$ ▷ P_I and N_I are the positive and negative individuals
  for all $ind \in P_I$ **do**
    Add $ind : dummyClass$ to $\mathcal{E}^+$ ▷ $\mathcal{E}^+$ the set of positive examples
  end for
  for all $ind \in N_I$ **do**
    Add $ind : dummyClass$ to $\mathcal{E}^-$ ▷ $\mathcal{E}^-$ is the set of negative examples
  end for
  $\mathcal{K}_{initial} = \mathcal{K}$
  for all $CE \in ClassExpressions$ **do**
    $Axiom = p :: CE \sqsubseteq dummyClass$
    $\mathcal{K}' = \mathcal{K} \cup \{Axiom\}$
    $(LL, \mathcal{K}') = \text{EDGE}(\mathcal{K}, \mathcal{E}^+, \mathcal{E}^-, \epsilon, \delta, NE, TLE)$
    if $LL > LL_0$ **then**
      $\mathcal{K} = \mathcal{K}'$
      Add $Axiom$ to $LearnedAxioms$
      $LL_0 = LL$
    end if
  end for
  if LP_{type} is $ClassLearning$ **then**
    for all $Axiom \in LearnedAxioms$ **do**
      $Axiom = CE \sqsubseteq TargetClass$
    end for
  end if
  $\mathcal{K} = \mathcal{K}_{initial} \cup LearnedAxioms$
end procedure

18.7 Related Work

GoldMiner [245, 246] is an algorithm that exploits Association Rules (ARs) for building ontologies. GoldMiner extracts information about individuals, named classes and roles using SPARQL queries. From these data, it builds two *transaction tables*: one that stores the classes to which each individual belongs and one that stores the roles to which each couple of individuals belongs. Finally, the APRIORI algorithm [247] is applied to each table in order to find ARs. Implications of the form $A \Rightarrow B$ can be converted to subclass axioms of the form $A \sqsubseteq B$. Moreover, the confidence p of an AR can be interpreted as the probability of the axiom $p :: A \sqsubseteq B$. So GoldMiner can be used to obtain a probabilistic knowledge base.

The structure learner LEAP is inspired to SLIPCOVER, an algorithm proposed for learning probabilistic logic programs based on distribution semantics [5]. LEAP shares with it the search strategy and the use of the log-likelihood of the data as the score of the learnt theories. Like SLIPCOVER, it divides the search between learning promising axioms and building in a greedy way a theory whose parameters are optimized by relying on a parameter learning algorithm.

A work that integrates parameter and structure learning for a probabilistic extension of $\mathcal{ALC}$, named CR$\mathcal{ALC}$, is [118]. CR$\mathcal{ALC}$ allows statistical axioms of the form $P(C|D) = \alpha$, meaning that for any element x in $\mathcal{D}$, the probability that it is in C given that is in D is α, and of the form $P(R) = \beta$, meaning that for each couple of elements x and y in $\mathcal{D}$, the probability that x is linked to y by the role R is β. CR$\mathcal{ALC}$ does not allow to express a degree of belief in axioms as DISPONTE.

An algorithm is presented in [118] that learns parameters and structure of CR$\mathcal{ALC}$ KBs. It starts from positive and negative examples for a single concept and from the general concept $\top$ in the root of a search tree to be refined. For a set of candidate concept definitions, their probabilistic parameters are learned using an EM algorithm and a score is assigned to the corresponding node. If the best score in the tree is above a threshold, a deterministic concept definition is returned, otherwise a probabilistic inclusion C_i is searched on a weighted spanning tree, where the target concept is added as a parent of each vertex and probabilities are learned as $P(C_i|Parents(C_i))$. We share the top-down procedure for building axioms (CELOE) but we exploit the BDD structures instead of resorting to inference in a graphical model to compute the expected counts for EM.

The paper [248] presents a statistical relational learning system for learning terminological naïve Bayesian classifiers, which estimate the probability that an individual a belongs to a certain target concept given its membership to a set of induced DL (feature) concepts. The classifier consists of a Bayesian Network (BN) modelling the dependency relations between the feature concepts and the target one. The learning process handles three different assumptions that can be made about the lack of knowledge (under OWA) regarding concept-membership, reflecting in the adoption of different scoring functions and search strategies of the optimal network and parameters. Under one of the assumptions - the probability of concept-membership of a depends on the knowledge on a available in $\mathcal{K}$ - the EM method is proposed to train the BN parameters. The classifier can be seen as a learner of probabilistic assertional axioms, while LEAP learns probabilistic terminological axioms. We exploit BDDs instead of

BNs, while we share with them the use of EM.

18.8 Experiments

LEAP has been evaluated on three KBs:

- Carcinogenesis[7] [227] describing the carcinogenicity of more than 300 chemical compounds. It contains 22,372 individuals and 74,409 axioms.
- The SoftWiki Ontology for Requirements Engineering (SWORE) [249] defining core concepts of requirements engineering and the way they are interrelated. It contains 107 individuals and 926 axioms.
- The Moral[8] KB that qualitatively simulates moral reasoning. It contains 202 individuals and 4710 axioms.

Regarding Carcinogenesis, we randomly selected 180 individuals, 103 of which representing positive examples for the class *Compound*, i.e. individuals that belong to the class *Compound*, and 77 representing negative examples, i.e. individuals that do not belong to the class *Compound*. For SWORE, we used all the 5 individuals that belong to the class *CustomerRequirement* as positive examples and 30 representing negative examples. For the Moral KB we selected all the 24 individuals for the class *Vicarious* as positive examples and 175 individuals randomly selected among the remaining ones as negative examples.

In the training phase, we first assigned a random probability to every axiom of the KB and we applied a 5-fold cross validation. We ran EDGE on the original KBs for learning the parameters associated with the probabilistic axioms, with $NE = 3$ and $TLE = \infty$ for the call to BUNDLE (cf. Alg. 16.1) in order to limit the runtime. Then, we separately ran LEAP on the original KBs for learning probabilistic subsumption axioms and the associated parameters for the class: *Compound* for Carcinogenesis KB, for which LEAP learned 1 axiom in every fold; *CustomerRequirement* for SWORE, for which LEAP learned 1 axiom in every fold and *Vicarious* for the Moral KB, where LEAP learned 9 axioms in three folds and 8 axioms in the others.

For CELOE, we set LP_{type} = *Positive and Negative Examples Learning Problem*, for Carcinogenesis we set $NC = 3$ while for the others we set $NC = 10$ and timeout TLC for its execution of 120 seconds: when the timeout expires or the maximum number of class expressions are found, the current set of them is returned to the caller.

In the testing phase, we computed the probability of the examples (queries) in the test set according to the KBs learned by LEAP and the original ones, by applying BUNDLE. We drew the PR and ROC curves and computed the AUCPR and AUCROC. Table 18.1 shows the AUCPR and the AUCROC averaged over the folds together with the standard deviation for all the KBs.

Most of the learning time was spent for building the BDDs of the examples. For instance, for the Carcinogenesis KB, on a total learning time of about 1,905 seconds,

[7] `http://dl-learner.org/wiki/Carcinogenesis`

[8] `https://archive.ics.uci.edu/ml/datasets/Moral+Reasoner`

Table 18.1: Results of the experiments in terms of AUCPR and AUCROC averaged over the folds with EDGE and LEAP. The first column shows the areas computed w.r.t. the resulting KB after the execution of EDGE. Standard deviations are also shown.

	EDGE		LEAP	
	AUCPR	AUCROC	AUCPR	AUCROC
Carcinogenesis	0.534 ± 0.108	0.445 ± 0.051	0.801 ± 0.240	0.798 ± 0.246
SWORE	0.148 ± 0.063	0.453 ± 0.272	1 ± 0	1 ± 0
Moral	0.119 ± 0.009	0.5 ± 0	1 ± 0	1 ± 0

only 139 seconds was used by CELOE, while 1,765 seconds was used for building BDDs. Only 0.206 seconds was spent for the initialization of the systems.

The p-value of a paired two-tailed t-test of the difference in AUCPR and AUCROC between the LEAP ontologies and the initial ones is 0.0603 and 0.0360 respectively for Carcinogenesis, $7.143 \cdot 10^{-6}$ and 0.0109 for SWORE, and $2.734 \cdot 10^{-9}$ and 0 for Moral. The results show that LEAP is useful in achieving better areas under both the PR and ROC curves, with statistically significant difference at the 5% significance level except for AUCPR on Carcinogenesis.

18.9 Conclusions

LEAP learns the structure on DISPONTE KBs by first performing a search in the space of promising axioms, by exploiting CELOE to learn class expressions of target concepts, and then a greedy search in the space of the ontologies. In this second phase the probabilities of the new axioms are computed by EDGE. The experiments show that LEAP achieves larger areas under both the PR and the ROC curve than a single execution of EDGE.

In the next chapter we show how we adapted LEAP to run with EDGE$^{\mathrm{MR}}$.

Chapter 19

Distributed Stucture Learning in Probabilistic Description Logics

Similarly to SLIPCOVER, LEAP suffers from scalability problems. In order to reduce the structure learning time we adapted LEAP to work with EDGE^{MR}. The resulting system was called LEAP^{MR}. The details of LEAP^{MR} are explained in Section 19.1. Section 19.2 shows a preliminary test performed to evaluate LEAP^{MR}. Section 19.3 concludes the chapter.

19.1 Distributed Structure Learning: LEAP^{MR}

LEAP^{MR} is an evolution of the LEAP system [16] presented in Chapter 18. While the latter exploits EDGE, the former was adapted to be able to perform EDGE^{MR}.

It performs structure and parameter learning of probabilistic ontologies under DISPONTE by exploiting: (1) CELOE [239] for the structure, and (2) EDGE^{MR} (Chapter 17) for the parameters. Figure 19.1 shows the architecture of LEAP^{MR}.

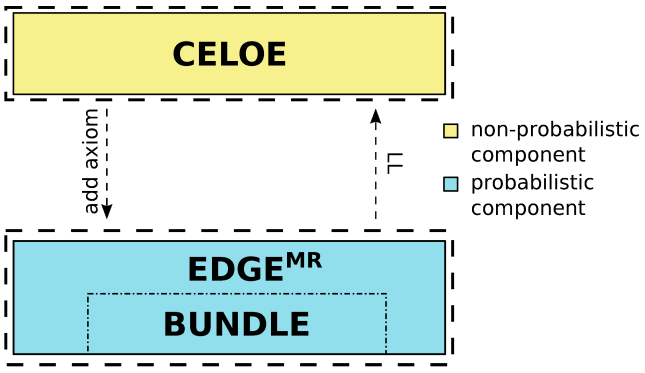


Figure 19.1: LEAP^{MR} architecture.

In order to learn an ontology, LEAP^{MR} first searches for good candidate probabilistic subsumption axioms by means of CELOE, then it performs a greedy search in the space of theories using EDGE^{MR} to evaluate the theories using the log-likelihood as heuristic.

Algorithm 19.1 shows LEAP$^{\mathrm{MR}}$'s main procedure: it takes as input the knowledge base $\mathcal{K}$ and the configuration settings for CELOE and EDGE$^{\mathrm{MR}}$, then generates $NumC$ class expressions by exploiting CELOE and the sets of positive and negative examples which will be the queries (concept membership axioms) for EDGE$^{\mathrm{MR}}$. Then LEAP$^{\mathrm{MR}}$ adds to $\mathcal{K}$ one probabilistic subsumption axiom generated from the class expression set at a time. After each addition, EDGE$^{\mathrm{MR}}$ is performed on the extended KB to compute the LL of the data and the parameters. If the LL is better than the current best, the new axiom is kept in the knowledge base and the parameter of the probabilistic axiom are updated, otherwise the learned axiom is removed from the ontology and the previous parameters are restored. The final theory is obtained from the union of the initial ontology and the probabilistic axioms learned.

Algorithm 19.1 Function LEAP$^{\mathrm{MR}}$.

function LEAP$^{\mathrm{MR}}(\mathcal{K}, LP_{type}, NumC, \epsilon, \delta, S)$
  Input: a knowledge base $\mathcal{K}$
  Input: the type LP_{type} of learning problem
  Input: the maximum number of class expressions to find NC
  Input: the time limit for the inference for CELOE TLC
  Input: a threshold ϵ for the difference between LLs
  Input: a threshold δ for the fraction of the difference between LLs
  Input: the maximum number of explanations to find for each example NL
  Input: the time limit for the inference for each example TLE
  Input: the scheduling method S
  Output: the learned knowledge base $\mathcal{K}$
  $ClassExpressions$ = up to NL or until TLC is reached ▷ generated by CELOE
  $(P_I, N_I) =$ ExtractIndividuals(LP_{type}) ▷ LP_{type}: specifies how to extract (P_I, N_I)
  for all $ind \in P_I$ **do** ▷ P_I: set of positive individuals
    Add ind : `Target` to E^+ ▷ E^+: set of positive ex
  end for
  for all $ind \in N_I$ **do** ▷ N_I: set of negative individuals
    Add ind : `Target` to E^- ▷ E^-: set of negative examples
  end for
  $(LL, \mathcal{K}') = EDGE^{MR}(\mathcal{K}', E^+, E^-, \epsilon, \delta, NL, TLE, S)$ ▷
  for all $CE \in ClassExpressions$ **do**
    $Axiom = p :: CE \sqsubseteq$ `Target`
    $\mathcal{K}' = \mathcal{K} \cup \{Axiom\}$
    $(LL, \mathcal{K}') = EDGE^{MR}(\mathcal{K}', E^+, E^-, \epsilon, \delta, NL, TLE, S)$ ▷ Call to $EDGE^{MR}$
    if $LL > LL_0$ **then**
      $\mathcal{K} = \mathcal{K}'$
      $LL_0 = LL$
    end if
  end for
  return $\mathcal{K}$
end function

19.2 Experiments

In order to test how much the exploitation of EDGE$^{\mathrm{MR}}$ can improve the performances of LEAP$^{\mathrm{MR}}$, we did a preliminary test where we considered the Moral[1] KB which qualitatively simulates moral reasoning. It contains 202 individuals and 4710 axioms (22 axioms are probabilistic).

We performed the experiments on a cluster of 64-bit Linux machines with 8-cores Intel Haswell 2.40 GHz CPUs and 2 GB (max) memory allotted to Java per node. We allotted 1, 3, 5, 9 and 17 nodes, where the execution with 1 node corresponds to the

[1] `https://archive.ics.uci.edu/ml/datasets/Moral+Reasoner`

execution of LEAP, while for the other configurations we used the dynamic scheduling with chunks containing 3 queries. For each experiment 2 candidate probabilistic axioms are generated by using CELOE and a maximum of 3 explanations per query was set for EDGE^{MR}. Figure 19.2 shows the speedup obtained as a function of the number of machines (nodes). The speedup is the ratio of the running time of 1 worker to the one of n workers. We can note that the speedup is significant even if it is sublinear, showing that a certain amount of overhead (the resources, and thereby the time, spent for the MPI communications) is present.

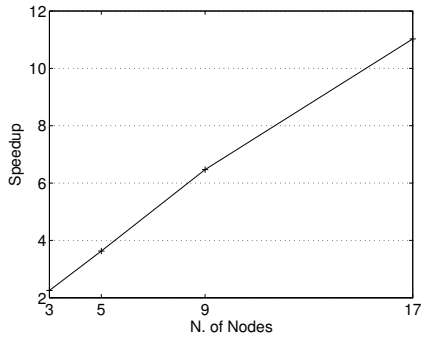


Figure 19.2: Speedup of LEAP^{MR} relative to LEAP for Moral KB.

19.3 Conclusion

This chapter presented the algorithm LEAP^{MR} for learning the structure of probabilistic description logics under DISPONTE. LEAP^{MR} performs EDGE^{MR} which is a MapReduce implementation of EDGE, exploiting modern computing infrastructures for performing distributed parameter learning.

As future work we would like to distribute both the structure and the parameter learning of probabilistic knowledge bases by exploiting EDGE^{MR} also during the building of the class expressions. In particular, we would like to distribute the scoring function used to evaluate the obtained refinements. In this function EDGE^{MR} take as input a KB containing only the individuals and the class expression to test. Finally, the class expressions found are sorted according to the LL returned by EDGE^{MR} and their initial probability are the probability learned during the execution of EDGE^{MR}.

With this chapter we conclude the part of this thesis dedicated to learning. The major issue of the proposed systems is that they do not scale well. In the next chapters we provide some final remarks of the work done and we discuss some ideas for future work.

Part V

Conclusions and Future Work

Chapter 20

Conclusions

The Distribution Semantics has provided a way to combine logic programs with probability theory. The logical languages that use the distributions semantics are called Probabilistic Logic Programming (PLP) languages. Among these we have LPADs, one of the formalisms used in this thesis.

With the advent of Semantic Web, that makes use of formalisms based on Description Logics (DLs) to represent knowledge, it has become increasingly important to have Probabilistic Description Logics. DISPONTE was developed to meet this requirement and applies the Distribution Semantics to Description Logics.

We worked both on Probabilistic Logic Programming (PLP) and Probabilistic Description Logic (PDL) research fields because we are convinced that the two areas are strictly intertwined, and that the advances achieved in one of them can improve the other.

The aim of this thesis was to provide inference and learning systems for uncertain relational data expressed in a probabilistic logical formalism such as LPAD and PDLs that follow DISPONTE.

Once we defined our semantics and our formalisms in **Part II - Probabilistic Logics**, we divided our developed systems into two parts. **Part III - Inference in Probabilistic Logics** presented our inference systems for PLP and PDLs, whereas **Part IV - Learning** presented our learning systems.

Inference

The proposed systems can be split into two main categories.

PLP Inference Systems `cplint` is a system that allows to perform approximate and exact inference on LPADs even if the LPAD contains predicates whose arguments are continuous random variables. Moreover `cplint` supports causality when the model is fully known. `cplint` on SWISH is a web application that allows to write and test LPADs without installing anything in the local machine. The causality feature was tested on random social networks of increasing size and compared to conditional inference. Surprisingly, the results show that exact probabilistic inference is faster than the approximate one.

PDL Inference Systems All the proposed systems perform probabilistic logic-based inference on PDLs that follow DISPONTE. The system BUNDLE, written in Java, resolves the MIN-A-ENUM problem for a given query on $\mathcal{SROIQ}(\mathbf{D})$ DISPONTE KBs and then builds the corresponding BDD. TRILL, instead, implements the tableau algorithm in Prolog. TRILL is limited to $\mathcal{SHIQ}(\mathbf{D})$ KBs. TRILL showed that a Prolog implementation is possible. Therefore we implemented a third reasoner called TRILL^P. This system instead of computing all the explanations for a given query, it computes the *pinpointing formula* by exploiting a SAT solver to test ψ-*insertability*. Since the explanations may grow exponentially, the complexity of all these systems is high. Moreover the computation of the probability through Binary Decision Diagrams has a #P-complexity in the number of explanations. Nevertheless, experiments applied on a real world datasets proved that these systems handle domains of significant size.

Learning

In the field of Statistical Relational Learning (SRL), the two main learning tasks are the *parameter learning*, we know the structure (the logic formulas) of the KB but we want to know the parameters (weights) of the logic formulas, and *structure learning* where both the structure and the parameters have to be learned.

The proposed systems tackles both the learning tasks. Here again, we can divide the developed learning systems into two categories.

PLP Learning Systems We presented an algorithm for structure learning of probabilistic logic programs, SEMPRE. This algorithm is a MapReduce implementation of SLIPCOVER. SEMPRE relies on a distributed parallel algorithm named EMBLEM$^{\text{MR}}$, which, in turn, is a MapReduce version of EMBLEM, a parameter learning algorithm for probabilistic logic programs.

PDL Learning Systems We have presented the EDGE system (Chapter 16), a supervised parameter learning algorithm which learns probability parameters of DISPONTE KBs by exploiting an Expectation Maximization algorithm executed over the BDDs built using BUNDLE. Starting from these results, we developed LEAP (Chapter 18), a supervised learning system able to learn both the structure and the parameters of a DL KB. LEAP exploits CELOE for creating candidate axioms and EDGE to both test the quality of the candidate axioms and learn the parameters of the resulting KB. The experiments showed that LEAP can achieve better results than simply tuning the parameters of an existing KB by using EDGE. All these algorithms are sequential and are rather expensive from a computational point of view. The diffusion of *Big Data* and the increased importance of *Linked Open Data* motivated us to develop parallel and distributed algorithms in order to manage huge amount of data. EDGE$^{\text{MR}}$ is a distributed version of EDGE based on the MapReduce approach (Chapter 17). It distributes the computational tasks between different workers. In particular, the examples' BDD building and the expectation step are split among the workers which run in parallel. We can have two different scheduling techniques: single-step and

dynamic. We exploit the Message Passing Interface (MPI) standard for communication. Finally the system LEAP$^{\mathrm{MR}}$ exploits EDGE$^{\mathrm{MR}}$ to speed up the learning time during the parameter learning phase (Chapter 19).

In our distributed algorithms, we did not use conventional MapReduce frameworks such as Hadoop, since they commonly require purely functional operations, while we have to deal with operations that have to keep part of information in memory.

The experimental results of our distributed systems showed that distribution is beneficial to effectively reduce the learning time.

Chapter 21

Future Work

In previous chapters we often mentioned some improvements that can be applied to our systems. In this chapter we concentrate and briefly analyse all the ideas for improvement scattered all over the thesis. Moreover, we present some ideas and concepts for possibly new systems. We can divide our future work in two folds, one dedicated to future work on inference (Section 21.1) and one to learning (Section 21.2).

21.1 Future Work on Inference

`cplint`
`cplint` causality feature can be extended to support Pearl's full *do* calculus. In fact we assume that the causal structure of the model is fully known. Pearl's *do* calculus, instead, is more general, as it allows to compute the effect of actions also on models with unknown variables. Exploiting the full power of the *do* calculus in PLP is a very interesting direction for future work.

BUNDLE
The current version of BUNDLE uses Pellet as reasoner under the hood, during the HST algorithm. However, the new versions of Pellet are no more open source. We are currently working to re-engineer BUNDLE in order to modularize it and make it work with other OWL reasoners. The OWLExplanation library[1] seems useful to reach this purpose, although the original fork has not been updated for two years. Moreover we are thinking of developing a web inference for BUNDLE integrated with WebProtégé [250] and a BUNDLE plugin for the ontology editor Protégé [76].

TRILL
TRILL currently supports $\mathcal{SHIQ}(\mathbf{D})$ KBs. It can be extended by adding the expansion rules needed to reason over $\mathcal{SROIQ}(\mathbf{D})$ KBs.

TRILL on SWISH
Currently in TRILL on SWISH we can write KBs only in OWL RDF/XML format.

[1] https://github.com/matthewhorridge/owlexplanation

To overcome this limit we are planning to integrate TRILL on SWISH with WebProtégé [250].

New Systems and Ideas
Some ideas and new systems for the future are:

- A new reasoner for approximate inference for DISPONTE KBs, that performs inference by sampling, can be developed by using the same techniques mentioned in Subsection 12.2.2.
- Allow our systems to automatically retrieve information on-line via public endpoints, such as SPARQL servers;

21.2 Future Work on Learning

EDGE
At present, EDGE depends on BUNDLE for obtaining the covering set of explanations. In the future, we plan to develop a reengineered version of EDGE such that it is independent of BUNDLE and can be used with other (probabilistic) OWL reasoners that returns the explanations such as TRILL.

LEAP and LEAP$^{\mathbf{MR}}$
We can optimize these systems as follows.

- Improving the scalability of our algorithms, in order to handle larger datasets. In particular we can think to use approaches for knowledge fragment selection [251, 252], i.e. extracting only the relevant part of the knowledge, in order to reduce the reasoning and hence the learning time.
- Exploiting scraping methods [253] to enrich our initial knowledge base and improve the learning.
- For LEAP$^{\mathrm{MR}}$ we could think to exploit EDGE$^{\mathrm{MR}}$ also when building class expressions. Similarly to SEMPRE [254, 13] (Section 15.5), we would like to distribute the scoring function used to evaluate the obtained refinements. In this function EDGE$^{\mathrm{MR}}$ takes as input a KB containing only the individuals and the class expression to test. Finally, the class expressions found are sorted according to the LL returned by EDGE$^{\mathrm{MR}}$ and their initial probability are the probability learned during the execution of EDGE$^{\mathrm{MR}}$. Currently LEAP and LEAP$^{\mathrm{MR}}$ support only supervised learning, we plan to add semi-supervised or unsupervised learning. Another branch of research is to adapt LEAP$^{\mathrm{MR}}$ to exploit Apache Spark and to run the queries on GPUs.

New Systems and Ideas
Concerning the development of new learning algorithms we can take two different directions. One based on purely symbolic SRL approaches and one that tries to combine probabilistic logics with neural networks.

Purely symbolic Structure Learning in PDL can be limited by DL expressivity. In fact, in order to reach decidability, there are severe restrictions on the way variables and quantifiers can be used. For instance in OWL it is impossible to write rules as the one in the following example (from [255]).

Example 21.2.1
In DL it is impossible to assert that individuals who live and work at the same location are "Home workers". With logic programs instead it is easy to represent that. For instance in Prolog:

```
home_worker(X) :- work(X,Y), live(X,Z), located(Y,W),
   located(Z,W).
```

These expressivity restrictions of DLs limit the rules that can be learned by structure learning algorithms. A way to avoid this problem is to map DLs (some specific fragments) to logic programs without losing the semantics, as proposed in [255] and [193]. Then we could think to use PLP learning algorithms such as SLIPCOVER or SEMPRE to build rules on top of DISPONTE KBs.

Neural-symbolic integration The success of neural network technologies is undeniable. There are countless academic and industrial applications. In these years several approaches which combine artificial neural networks with symbolic logical representation techniques have emerged. For knowledge extraction from trained neural networks [256, 257, 258] and for learning new RDF triples by means of entity latent features [259]. A parallel thread of our research will concern a study of these methods.

Bibliography

[1] T. Sato. "A Statistical Learning Method for Logic Programs with Distribution Semantics". In: *12th International Conference on Logic Programming, Tokyo, Japan*. Ed. by L. Sterling. Cambridge, Massachusetts: MIT Press, 1995, pp. 715–729. ISBN: 0-262-69177-9.

[2] E. Bellodi, E. Lamma, F. Riguzzi, and S. Albani. "A Distribution Semantics for Probabilistic Ontologies". In: *7th International Workshop on Uncertainty Reasoning for the Semantic Web*. Vol. 778. CEUR Workshop Proceedings. Aachen, Germany: Sun SITE Central Europe, 2011, pp. 75–86.

[3] F. Riguzzi, E. Bellodi, E. Lamma, and R. Zese. "Probabilistic Description Logics under the Distribution Semantics". In: *Semantic Web – Interoperability, Usability, Applicability* 6.5 (2015), pp. 447–501. DOI: 10.3233/SW-140154.

[4] E. Bellodi and F. Riguzzi. "Expectation Maximization over Binary Decision Diagrams for Probabilistic Logic Programs". In: *Intelligent Data Analysis* 17.2 (2013), pp. 343–363.

[5] E. Bellodi and F. Riguzzi. "Structure Learning of Probabilistic Logic Programs by Searching the Clause Space". In: *Theory and Practice of Logic Programming* 15.2 (2015), pp. 169–212. DOI: 10.1017/S1471068413000689.

[6] F. Riguzzi, G. Cota, E. Bellodi, and R. Zese. "Causal inference in cplint". In: *International Journal of Approximate Reasoning* 91 (2017), pp. 216–232. ISSN: 0888-613X. DOI: https://doi.org/10.1016/j.ijar.2017.09.007. URL: https://www.sciencedirect.com/science/article/pii/S0888613X17301640.

[7] M. Alberti, E. Bellodi, G. Cota, F. Riguzzi, and R. Zese. "`cplint` on SWISH: Probabilistic Logical Inference with a Web Browser". In: *Intelligenza Artificiale* 11.1 (2017), pp. 47–64. DOI: 10.3233/IA-170105.

[8] F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. "Probabilistic Logic Programming on the Web". In: *Software: Practice and Experience* 46.10 (Oct. 2016), pp. 1381–1396. DOI: 10.1002/spe.2386.

[9] F. Riguzzi and G. Cota. *Probabilistic Logic Programming Tutorial*. London, UK, Apr. 2016. URL: http://www.cs.nmsu.edu/ALP/2016/03/probabilistic-logic-programming-tutorial/.

[10] R. Zese, E. Bellodi, F. Riguzzi, G. Cota, and E. Lamma. "Tableau Reasoning for Description Logics and its Extension to Probabilities". In: *Annals of Mathematics and Artificial Intelligence* (2016), pp. 1–30. DOI: 10.1007/s10472-016-9529-3. URL: http://dx.doi.org/10.1007/s10472-016-9529-3f.

[11] F. Riguzzi, E. Bellodi, E. Lamma, and R. Zese. "Reasoning with Probabilistic Ontologies". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence, Buenos Aires, Argentina*. Ed. by Q. Yang and M. Wooldridge. Palo Alto, California USA: AAAI Press/International Joint Conferences on Artificial Intelligence, 2015, pp. 4310–4316. ISBN: 978-1-57735-738-4.

[12] E. Bellodi, E. Lamma, F. Riguzzi, R. Zese, and G. Cota. "A web system for reasoning with probabilistic OWL". In: *Software: Practice and Experience* 47.1 (2017), pp. 125–142.

[13] F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. "Scaling Structure Learning of Probabilistic Logic Programs by MapReduce". In: *Proceedings of the 22nd European Conference on Artificial Intelligence*. Ed. by M. Fox and G. Kaminka. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 1602–1603. DOI: 10.3233/978-1-61499-672-9-1602.

[14] F. Riguzzi, E. Bellodi, E. Lamma, and R. Zese. "Parameter Learning for Probabilistic Ontologies". In: *7th International Conference on Web Reasoning and Rule Systems (RR 2013), Mannheim, Germany*. Ed. by W. Faber and D. Lembo. Vol. 7994. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 265–270. ISBN: 978-3-642-39665-6. DOI: 10.1007/978-3-642-39666-3_26.

[15] G. Cota, R. Zese, E. Bellodi, F. Riguzzi, and E. Lamma. "Distributed Parameter Learning for Probabilistic Ontologies". In: *25th International Conference on Inductive Logic Programming*. Ed. by K. Inoue, H. Ohwada, and A. Yamamoto. 2015.

[16] F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. "Learning Probabilistic Description Logics". English. In: *Uncertainty Reasoning for the Semantic Web III*. Ed. by F. Bobillo et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer International Publishing, 2014, pp. 63–78. ISBN: 978-3-319-13412-3. DOI: 10.1007/978-3-319-13413-0_4.

[17] G. Cota, R. Zese, E. Bellodi, E. Lamma, and F. Riguzzi. "Structure Learning with Distributed Parameter Learning for Probabilistic Ontologies". In: *Doctoral Consortium of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2015)*. Ed. by J. Hollmen and P. Papapetrou. 2015, pp. 75–84. ISBN: 978-952-60-6443-7. URL: http://urn.fi/URN:ISBN:978-952-60-6443-7.

[18] R. A. Kowalski. "Predicate Logic as Programming Language". In: *IFIP Congress*. 1974, pp. 569–574.

[19] M. H. Van Emden and R. A. Kowalski. "The semantics of predicate logic as a programming language". In: *Journal of the ACM* 23.4 (1976), pp. 733–742.

[20] R. Reiter. "On closed world data bases". In: *Logic and Data Bases*. Plenum Press, 1978, pp. 55–76.

[21] K. L. Clark. "Negation as failure". In: *Logic and data bases*. Springer, 1978, pp. 293–322.

[22] M. Gelfond and V. Lifschitz. "The stable model semantics for logic programming." In: *Logic Programming, 5th International Conference and Symposium, Seattle, Washington*. Vol. 88. MIT Press, 1988, pp. 1070–1080.

[23] T. C. Przymusinski. "Every Logic Program Has a Natural Stratification And an Iterated Least Fixed Point Model". In: *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-1989)*. ACM Press, 1989, pp. 11–21.

[24] A. Van Gelder, K. A. Ross, and J. S. Schlipf. "The Well-founded Semantics for General Logic Programs". In: *J. ACM* 38.3 (1991), pp. 620–650.

[25] K. R. Apt and M. Bezem. "Acyclic Programs". In: *New generation computing* 9.3-4 (1991), pp. 335–363.

[26] W. Faber, G. Pfeifer, and N. Leone. "Semantics and complexity of recursive aggregates in answer set programming". In: *Artificial Intelligence* 175.1 (2011), pp. 278–298.

[27] F. Riguzzi. "Extended Semantics and Inference for the Independent Choice Logic". In: *Logic Journal of the IGPL* 17.6 (2009), pp. 589–629. DOI: 10.1093/jigpal/jzp025.

[28] F. Fages. "Consistency of Clark's completion and existence of stable models". In: *Meth. of Logic in CS* 1.1 (1994), pp. 51–60.

[29] L. De Raedt and A. Kimmig. "Probabilistic (Logic) Programming Concepts". In: *Machine Learning* 100.1 (2015), pp. 5–47.

[30] D. Fierens et al. "Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas". In: *Theory and Practice of Logic Programming* 15.3 (2015), pp. 358–401.

[31] J. Vennekens, S. Verbaeten, and M. Bruynooghe. "Logic Programs With Annotated Disjunctions". In: *Logic Programming, 24th International Conference, ICLP 2004, Saint-Malo, France, Proceedings*. Ed. by B. Demoen and V. Lifschitz. Vol. 3131. Lecture Notes in Computer Science. Berlin Heidelberg, Germany: Springer, 2004, pp. 431–445. ISBN: 978-3-540-27775-0. DOI: 10.1007/978-3-540-27775-0_30.

[32] M. Richardson and P. Domingos. "Markov logic networks". In: *Machine Learning* 62.1-2 (2006), pp. 107–136.

[33] E. Dantsin. "Probabilistic Logic Programs and their Semantics". In: *Russian Conference on Logic Programming*. Vol. 592. LNCS. Springer, 1991, pp. 152–164.

[34] R. T. Ng and V. S. Subrahmanian. "Probabilistic Logic Programming". In: *Information and Computation* 101.2 (1992), pp. 150–201.

[35] D. Poole. "Logic Programming, Abduction and Probability - A Top-Down Anytime Algorithm for Estimating Prior and Posterior Probabilities". In: *New Gen. Comp.* 11.3 (1993), pp. 377–400.

[36] M. P. Wellman, J. S. Breese, and R. P. Goldman. "From knowledge bases to decision models". In: *The Knowledge Engineering Review* 7.1 (1992), pp. 35–53.

[37] F. Bacchus. "Using First-Order Probability Logic for the Construction of Bayesian Networks". In: *9th Conference Conference on Uncertainty in Artificial Intelligence (UAI 1994)*. Morgan Kaufmann Publishers, 1993, pp. 219–219.

[38] L. Ngo and P. Haddawy. "Answering queries from context-sensitive probabilistic knowledge bases". In: *Theoretical Computer Science* 171.1 (1997), pp. 147–177.

[39] K. Kersting and L. De Raedt. "Towards Combining Inductive Logic Programming with Bayesian Networks". In: *11th International Conference on Inductive Logic Programming*. Springer, 2001, pp. 118–131.

[40] V. S. Costa, D. Page, M. Qazi, and J. Cussens. "CLP (BN): Constraint Logic Programming for Probabilistic Knowledge". In: *19th International Conference on Uncertainty in Artificial Intelligence (UAI-03)*. Morgan Kaufmann Publishers, 2003, pp. 517–524.

[41] T. Gomes and V. S. Costa. "Evaluating Inference Algorithms for the Prolog Factor Language". In: *22nd International Conference on Inductive Logic Programming*. Ed. by F. Riguzzi and F. Zelezný. Vol. 7842. LNCS. Springer, 2012, pp. 74–85.

[42] F. Riguzzi and T. Swift. "Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics". In: *Theory and Practice of Logic Programming* 13.2 (2013), pp. 279–302. DOI: 10.1017/S1471068411000664.

[43] F. Riguzzi. "The Distribution Semantics for Normal Programs with Funtion Symbols". In: *International Journal of Approximate Reasoning* 77 (2016), pp. 1–19.

[44] D. Poole. "The Independent Choice Logic for Modelling Multiple Agents under Uncertainty". In: *Artificial Intelligence* 94 (1997), pp. 7–56.

[45] L. De Raedt, A. Kimmig, and H. Toivonen. "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India (IJCAI-07)*. Ed. by M. M. Veloso. Vol. 7. Palo Alto, California USA: AAAI Press, 2007, pp. 2462–2467.

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

[47] C. Baral, M. Gelfond, and N. Rushton. "Probabilistic reasoning with answer sets". In: *Theory and Practice of Logic Programming* 9.1 (2009), pp. 57–144. DOI: 10.1017/S1471068408003645.

[48] J. Vennekens, M. Denecker, and M. Bruynooghe. "CP-logic: A language of causal probabilistic events and its relation to logic programming". In: *Theory and Practice of Logic Programming* 9.3 (2009), pp. 245–308.

[49] F. Riguzzi. "MCINTYRE: A Monte Carlo System for Probabilistic Logic Programming". In: *Fundamenta Informaticae* 124.4 (2013), pp. 521–541. DOI: 10.3233/FI-2013-847.

[50] F. Riguzzi. "The Distribution Semantics for Normal Programs with Function Symbols". In: *International Journal of Approximate Reasoning* 77 (2016), pp. 1 –19. DOI: 10.1016/j.ijar.2016.05.005.

[51] L. De Raedt et al. "Towards digesting the alphabet-soup of statistical relational learning". In: *NIPS*2008 Workshop on Probabilistic Programming*. 2008.

[52] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0-521-78176-0.

[53] F. Baader, I. Horrocks, and U. Sattler. "Description Logics". In: *Handbook of Knowledge Representation*. Amsterdam: Elsevier, 2008. Chap. 3, pp. 135–179. ISBN: 978-0-444-52211-5.

[54] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[55] W3C. *OWL 2 Web Ontology Language*. Dec. 2012. URL: http://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[56] W3C. *OWL 2 web ontology language: Structural specification and functional-style syntax*. Dec. 2012. URL: https://www.w3.org/TR/owl2-syntax/.

[57] G. Chapman, J. Cleese, T. Gilliam, E. Idle, T. Jones, and M. Palin. *Monty Python and the Holy Grail*. 1975.

[58] F. Baader and U. Sattler. "Number Restrictions on Complex Roles in Description Logics". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference*. 1996, pp. 328–339.

[59] I. Horrocks, U. Sattler, and S. Tobies. "Practical reasoning for expressive description logics". In: *Logic for Programming and Automated Reasoning*. Vol. 99. Springer. 1999, pp. 161–180.

[60] I. Horrocks and U. Sattler. "A description logic with transitive and inverse roles and role hierarchies". In: *Journal of Logic and Computation* 9.3 (1999), pp. 385–410.

[61] I. Horrocks, O. Kutz, and U. Sattler. "The Even More Irresistible $\mathcal{SROIQ}$". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Tenth International Conference*. Vol. 6. Lake District, UK: AAAI Press, 2006, pp. 57–67. ISBN: 978-1-57735-271-6. URL: http://dl.acm.org/citation.cfm?id=3029947.3029959.

[62] Y. Kazakov. "$\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference*. KR'08. Sydney, Australia: AAAI Press, 2008, pp. 274–284. ISBN: 978-1-57735-384-3. URL: https://www.cs.ox.ac.uk/files/362/Kazakov_KR08_RIQ_SROIQ.pdf.

[63] A. Borgida. "On the relative expressiveness of description logics and predicate logics". In: *Artificial Intelligence* 82.1-2 (1996), pp. 353–367.

[64] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. "Description Logic Programs: Combining Logic Programs with Description Logic". In: *Proceedings of the12th International Conference on World Wide Web*. WWW '03. Budapest, Hungary: ACM Press, 2003, pp. 48–57. ISBN: 1-58113-680-3. DOI: 10.1145/775152.775160. URL: http://doi.acm.org/10.1145/775152.775160.

[65] W3C. *Semantic Web Activity*. 2001. URL: http://www.w3.org/2001/sw/.

[66] W3C OWL Working Group. *OWL Web Ontology Language*. 2004. URL: http://www.w3.org/TR/owl-features/.

[67] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. "OWL 2: The next step for OWL". In: *Journal of Web Semantics* 6.4 (2008), pp. 309–322.

[68] W3C OWL Working Group. *OWL 2 Web Ontology Language New Features and Rationale (Second Edition)*. Dec. 2012. URL: https://www.w3.org/TR/owl2-new-features/.

[69] W3C. *RDF/XML Syntax Specification*. 2004. URL: http://www.w3.org/TR/REC-rdf-syntax/.

[70] W3C. *Turtle - Terse RDF Triple Language*. 2011. URL: http://www.w3.org/TeamSubmission/turtle/.

[71] M. Horridge, N. Drummond, J. Goodwin, A. Rector, R. Stevens, and H. Wang. "The Manchester OWL Syntax". In: *OWLED2006 Second Workshop on OWL Experiences and Directions*. 2006.

[72] W3C. *OWL 2 Web Ontology Language Manchester Syntax (Second Edition)*. 2012. URL: http://www.w3.org/TR/owl2-manchester-syntax/.

[73] W3C. *OWL 1.1 Web Ontology Language: Structural Specification and Functional-Style Syntax*. 2008. URL: http://www.w3.org/TR/owl11-syntax/.

[74] W3C. *OWL 2 Web Ontology Language XML Serialization (Second Edition)*. 2012. URL: http://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211/.

[75] W3C. *OWL 2 Web Ontology Language: Profiles*. Dec. 2012. URL: https://www.w3.org/TR/owl2-profiles/.

[76] University of Stanford. *Protégé*. URL: http://protege.stanford.edu.

[77] MIND lab at University of Maryland. *Swoop*. URL: https://code.google.com/p/swoop/.

[78] R. Shearer, B. Motik, and I. Horrocks. "HermiT: A Highly-Efficient OWL Reasoner." In: *OWL: Experiences and Direction*. Vol. 432. 2008, p. 91.

[79] University of Oxford. *Hermit*. URL: http://hermit-reasoner.com/.

[80] *FaCT++*. URL: https://code.google.com/p/factplusplus/.

[81] D. Tsarkov and I. Horrocks. "FaCT++ Description Logic Reasoner: System Description". In: IJCAR'06 (2006), pp. 292–297. DOI: 10.1007/11814771_26. URL: http://dx.doi.org/10.1007/11814771_26.

[82] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, and Y. Katz. "Pellet: A practical OWL-DL reasoner". In: *Journal of Web Semantics* 5.2 (2007), pp. 51–53.

[83] M. Horridge and S. Bechhofer. "The owl api: A java api for owl ontologies". In: *Semantic Web* 2.1 (2011), pp. 11–21.

[84] *OWL API*. URL: http://owlapi.sourceforge.net/.

[85] A. S. Foundation. *Jena*. URL: http://jena.apache.org/.

[86] D. Calvanese, G. De Giacomo, and M. Lenzerini. "Reasoning in Expressive Description Logics with Fixpoints Based on Automata on Infinite Trees". In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence, Stockholm, Sweden (IJCAI-99)*. IJCAI'99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., 1999, pp. 84–89. URL: http://dl.acm.org/citation.cfm?id=1624218.1624231.

[87] U. Hustadt and R. A. Schmidt. "Using resolution for testing modal satisfiability and building models". In: *Journal of Automated Reasoning* 28.2 (2002), pp. 205–232.

[88] U. Hustadt, B. Motik, and U. Sattler. "Deciding expressive description logics in the framework of resolution". In: *Information and Computation* 206.5 (2008), pp. 579–601.

[89] F. Baader, S. Brandt, and C. Lutz. "Pushing the EL envelope". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence, Edinburgh, Scotland (IJCAI-05)*. Vol. 5. Morgan Kaufmann Publishers, 2005, pp. 364–369.

[90] A. Kalyanpur. "Debugging and Repair of OWL Ontologies". PhD thesis. The Graduate School of the University of Maryland, 2006.

[91] E. Bellodi. "Integration of Logic and Probability in Terminological and Inductive Reasoning". PhD thesis. Università degli Studi di Ferrara, 2013.

[92] R. Zese. *Probabilistic Semantic Web: Reasoning and Learning*. Vol. 28. Studies on the Semantic Web. Amsterdam: IOS Press, 2017. ISBN: 978-1-61499-733-7. DOI: 10.3233/978-1-61499-734-4-i. URL: http://ebooks.iospress.nl/volume/probabilistic-semantic-web-reasoning-and-learning.

[93] I. Horrocks and U. Sattler. "A Tableau Decision Procedure for $\mathcal{SHOIQ}$". In: *Journal of Automated Reasoning* 39.3 (2007), pp. 249–276.

[94] C. Halaschek-Wiener, A. Kalyanpur, and B. Parsia. *Extending Tableau Tracing for ABox Updates*. Tech. rep. University of Maryland, 2006.

[95] A. Kalyanpur, B. Parsia, M. Horridge, and E. Sirin. "Finding All Justifications of OWL DL Entailments". In: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*. Ed. by K. Aberer et al. Vol. 4825. Lecture Notes in Computer Science (LNCS). Berlin: Springer, 2007, pp. 267–280.

[96] A. Kalyanpur, B. Parsia, E. Sirin, and J. A. Hendler. "Debugging unsatisfiable classes in OWL ontologies". In: *Journal of Web Semantics* 3.4 (2005), pp. 268–293.

[97] S. Schlobach and R. Cornet. "Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies". In: *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Ed. by G. Gottlob and T. Walsh. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 355–362.

[98] A. Kalyanpur, B. Parsia, B. Cuenca-Grau, and E. Sirin. *Kalyanpur, Aditya and Parsia, Bijan and Cuenca-Grau, Bernardo and Sirin, Evren*. Tech. rep. 2005-66. University of Maryland, 2005.

[99] R. Reiter. "A Theory of Diagnosis from First Principles". In: *Artificial Intelligence* 32.1 (1987), pp. 57–95.

[100] F. Baader and R. Peñaloza. "Automata-Based Axiom Pinpointing". In: *Journal of Automated Reasoning* 45.2 (2010), pp. 91–129.

[101] F. Baader and R. Peñaloza. "Axiom Pinpointing in General Tableaux". In: *Journal of Logic and Computation* 20.1 (2010), pp. 5–34.

[102] F. Patel-Schneider P, I. Horrocks, and S. Bechhofer. *Tutorial on OWL*. 2003. URL: `http://www.cs.man.ac.uk/~horrocks/ISWC2003/Tutorial/`.

[103] J. Y. Halpern. "An Analysis of First-Order Logics of Probability". In: *Artificial Intelligence* 46.3 (1990), pp. 311–350.

[104] URW3-XG. *Uncertainty Reasoning for the World Wide Web, Final report*. 2008.

[105] F. Bacchus. *Representing and reasoning with probabilistic knowledge - a logical approach to probabilities*. Cambridge, MA, USA: MIT Press, 1990, pp. 1–233.

[106] C. Lutz and L. Schröder. "Probabilistic Description Logics for Subjective Uncertainty". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. Ed. by F. Lin, U. Sattler, and M. Truszczynski. Menlo Park, CA, USA: AAAI Press, 2010, pp. 393–403.

[107] J. Heinsohn. "Probabilistic Description Logics". In: *Proceedings of the 10th Conference Conference on Uncertainty in Artificial Intelligence (UAI 1994), Jul 29-31 1994, Seattle, WA*. Ed. by R. L. de Mántaras and D. Poole. Morgan Kaufmann, 1994, pp. 311–318.

[108] M. Jaeger. "Probabilistic Reasoning in Terminological Logics". In: *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*. Ed. by J. Doyle, E. Sandewall, and P. Torasso. Morgan Kaufmann, 1994, pp. 305–316.

[109] P. C. G. Da Costa, K. B. Laskey, and K. J. Laskey. "PR-OWL: A Bayesian Ontology Language for the Semantic Web". In: *Proceedings of the 2005 International Conference on Uncertainty Reasoning for the Semantic Web - Volume 173*. Uncertainty Reasoning for the Semantic Web I. Galway, Ireland: CEUR-WS.org, 2005, pp. 23–33.

[110] R. N. Carvalho, K. B. Laskey, and P. C. G. Costa. "PR-OWL 2.0 - Bridging the gap to OWL semantics". In: *Uncertainty Reasoning for the Semantic Web II.* Ed. by F. Bobillo and et al. Vol. 654. CEUR Workshop Proceedings. Sun SITE Central Europe, 2010.

[111] K. B. Laskey and P. C. G. da Costa. "Of Starships and Klingons: Bayesian Logic for the 23rd Century". In: *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh (UAI 2005), Scotland, July 26-29, 2005.* AUAI Press, 2005, pp. 346–353.

[112] R. Giugno and T. Lukasiewicz. "P-SHOQ(D): A Probabilistic Extension of SHOQ(D) for Probabilistic Ontologies in the Semantic Web". In: *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, Proceedings.* Ed. by S. Flesca, S. Greco, N. Leone, and G. Ianni. Vol. 2424. Lecture Notes in Computer Science. Springer, 2002, pp. 86–97.

[113] T. Lukasiewicz. "Probabilistic Default Reasoning with Conditional Constraints". In: *Annals of Mathematics and Artificial Intelligence* 34.1-3 (2002), pp. 35–88.

[114] T. Lukasiewicz. "Expressive probabilistic description logics". In: *Artificial Intelligence* 172.6-7 (2008), pp. 852–883.

[115] A. Calì, T. Lukasiewicz, L. Predoiu, and H. Stuckenschmidt. "Tightly coupled probabilistic description logic programs for the Semantic Web". In: *Journal on Data Semantics XII* (2009), pp. 95–130.

[116] Z. Ding and Y. Peng. "A Probabilistic Extension to Ontology Language OWL". In: *37th Hawaii International Conference on System Sciences (HICSS-37 2004), CD-ROM / Abstracts Proceedings, 5-8 January 2004, Big Island, HI, USA.* IEEE Computer Society, 2004.

[117] D. Koller, A. Y. Levy, and A. Pfeffer. "P-CLASSIC: A Tractable Probablistic Description Logic". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.* Ed. by B. Kuipers and B. L. Webber. AAAI Press / The MIT Press, 1997, pp. 390–397.

[118] J. E. O. Luna, K. Revoredo, and F. G. Cozman. "Learning Probabilistic Description Logics: A Framework and Algorithms". In: *Advances in Artificial Intelligence - 10th Mexican International Conference on Artificial Intelligence, MICAI 2011, Puebla, Mexico, November 26 - December 4, 2011, Proceedings, Part I.* Ed. by I. Z. Batyrshin and G. Sidorov. Vol. 7094. Lecture Notes in Computer Science. Berlin: Springer, 2011, pp. 28–39.

[119] C. d'Amato, N. Fanizzi, and T. Lukasiewicz. "Tractable Reasoning with Bayesian Description Logics". In: *Scalable Uncertainty Management, Second International Conference, SUM 2008, Naples, Italy, October 1-3, 2008. Proceedings.* Ed. by S. Greco and T. Lukasiewicz. Vol. 5291. Lecture Notes in Computer Science. Berlin: Springer, 2008, pp. 146–159.

[120] G. Gottlob, T. Lukasiewicz, and G. I. Simari. "Conjunctive Query Answering in Probabilistic Datalog+/- Ontologies". In: *5th International Conference on Web Reasoning and Rule Systems (RR 2011), Galway, Ireland*. Ed. by S. Rudolph and C. Gutierrez. Vol. 6902. Lecture Notes in Computer Science. Berlin: Springer, 2011, pp. 77–92.

[121] F. Riguzzi and T. Swift. "Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions". In: *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*. Vol. 7. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 162–171. ISBN: 978-3-939897-17-0. DOI: 10.4230/LIPIcs.ICLP.2010.162.

[122] F. Riguzzi and T. Swift. "The PITA system: Tabling and answer subsumption for reasoning under uncertainty". In: *Theory and Practice of Logic Programming* 11.4-5 (2011), pp. 433–449.

[123] A. Thayse, M. Davio, and J. P. Deschamps. "Optimization of multivalued decision algorithms". In: *International Symposium on Multiple-Valued Logic*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1978, pp. 171–178.

[124] S. B. Akers. "Binary decision diagrams". In: *IEEE Transactions on Computers* 6 (1978), pp. 509–516.

[125] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340. DOI: 10.1109/TC.1986.1676819. URL: http://dx.doi.org/10.1109/TC.1986.1676819.

[126] R. E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams". In: *ACM Computing Surveys* 24.3 (Sept. 1992), pp. 293–318. ISSN: 0360-0300. DOI: 10.1145/136035.136043. URL: http://doi.acm.org/10.1145/136035.136043.

[127] B. Bollig and I. Wegener. "Improving the Variable Ordering of OBDDs Is NP-Complete". In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002.

[128] F. Somenz. *CUDD: CU Decision Diagram Package Release 3.0.0*. University of Colorado. 2015. URL: http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[129] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. "High performance BDD package by exploiting memory hierarchy". In: *Proceedings of the 33rd annual Design Automation Conference*. ACM Press, 1996, pp. 635–640.

[130] *BuDDy: A BDD package*. URL: http://buddy.sourceforge.net/manual/main.html.

[131] A. Vahidi. *JDD: a pure Java BDD and Z-BDD library*. https://bitbucket.org/vahidi/jdd. 2015.

[132] J. Whaley. *JavaBDD*. URL: http://javabdd.sourceforge.net/.

[133] D. Poole. "First-order Probabilistic Inference". In: *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Ed. by G. Gottlob and T. Walsh. Morgan Kaufmann Publishers, 2003, pp. 985–991.

[134] E. Bellodi, E. Lamma, F. Riguzzi, V. S. Costa, and R. Zese. "Lifted Variable Elimination for Probabilistic Logic Programming". In: *Theory and Practice of Logic Programming* 14.4-5 (2014), pp. 681–695. DOI: 10.1017/S1471068414000283.

[135] G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. De Raedt. "Lifted Probabilistic Inference by First-Order Knowledge Compilation". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. Ed. by T. Walsh. IJCAI/AAAI, 2011, pp. 2178–2185.

[136] G. Van den Broeck. "On the Completeness of First-Order Knowledge Compilation for Lifted Probabilistic Inference". In: *Advances in Neural Information Processing Systems*. 2011, pp. 1386–1394.

[137] G. Van den Broeck. "Lifted Inference and Learning in Statistical Relational Models". PhD thesis. Ph. D. Dissertation, KU Leuven, 2013.

[138] F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. "A Survey of Lifted Inference Approaches for Probabilistic Logic Programming under the Distribution Semantics". In: *International Journal of Approximate Reasoning* 80 (Jan. 2017), pp. 313–333. ISSN: 0888-613X. DOI: 10.1016/j.ijar.2016.10.002.

[139] K. Kersting. "Lifted probabilistic inference". In: *Proceedings of the 20th European Conference on Artificial Intelligence*. IOS Press. 2012, pp. 33–38.

[140] A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. "On the implementation of the probabilistic logic programming language ProbLog". In: *Theory and Practice of Logic Programming* 11.2-3 (2011), pp. 235–262.

[141] J. Renkens, A. Kimmig, G. Van den Broeck, and L. De Raedt. "Explanation-Based Approximate Weighted Model Counting for Probabilistic Logics". In: *Proceedings of the 13th AAAI Conference on Statistical Relational AI*. AAAIWS'14-13. AAAI Press, 2014, pp. 86–92.

[142] D. Poole. "Abducing through Negation as Failure: Stable models within the independent choice logic". In: *J. Logic Program.* 44.1-3 (2000), pp. 5–35.

[143] D. Poole. "Probabilistic Horn Abduction and Bayesian Networks". In: *Artificial Intelligence* 64.1 (1993), pp. 81–129.

[144] F. Riguzzi and T. Swift. "The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty". In: *Theory and Practice of Logic Programming* 11.4–5 (2011), pp. 433–449. DOI: 10.1017/S147106841100010X.

[145] D. Poole. *AILog User Manual Version 2.3*. 2008.

[146] L. G. Valiant. "The complexity of enumeration and reliability problems". In: *SIAM Journal on Computing* 8.3 (1979), pp. 410–421.

[147] A. Darwiche and P. Marquis. "A Knowledge Compilation Map". In: *Journal of Artificial Intelligence Research* 17 (2002), pp. 229–264.

[148] T. Sang, P. Beame, and H. A. Kautz. "Performing Bayesian Inference by Weighted Model Counting". In: *20th National Conference on Artificial Intelligence*. Palo Alto, California USA: AAAI Press, 2005, pp. 475–482.

[149] C. Meinel and A. Slobodová. "On the complexity of constructing optimal ordered binary decision diagrams". In: *Mathematical Foundations of Computer Science* (1994), pp. 515–524.

[150] T. Swift and D. S. Warren. "XSB: Extending Prolog with Tabled Logic Programming". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 157–187. DOI: 10.1017/S1471068411000500.

[151] V. Santos Costa, R. Rocha, and L. Damas. "The YAP Prolog system". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 5–34.

[152] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. DOI: 10.1017/S1471068411000494.

[153] J. Von Neumann. "Various Techniques Used in Connection With Random Digits". In: *Nat. Bureau Stand. Appl. Math. Ser.* 12 (1951), pp. 36–38.

[154] A. Nampally and C. Ramakrishnan. "Adaptive MCMC-Based Inference in Probabilistic Logic Programs". In: *arXiv preprint arXiv:1403.6036* (2014).

[155] J. Pearl. *Causality*. Cambridge University Press, 2000.

[156] S. Wright. "Correlation and causation". In: *Journal of agricultural research* 20.7 (1921), pp. 557–585.

[157] C. Holzbaur. "Metastructures vs. attributed variables in the context of extensible unification". In: *Programming Language Implementation and Logic Programming: 4th International Symposium, PLILP'92 Leuven, Belgium, August 26–28, 1992 Proceedings*. Ed. by M. Bruynooghe and M. Wirsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 260–268. ISBN: 978-3-540-47297-1. DOI: 10.1007/3-540-55844-6_141.

[158] G. Van den Broeck, I. Thon, M. van Otterlo, and L. De Raedt. "DTProbLog: A Decision-Theoretic Probabilistic Prolog". In: *24th AAAI Conference on Artificial Intelligence, AAAI'10, Atlanta, Georgia, USA, July 11-15, 2010*. Ed. by M. Fox and D. Poole. AAAI Press, 2010, pp. 1217–1222.

[159] A.-L. Barabási and R. Albert. "Emergence of scaling in random networks". In: *Science* 286.5439 (1999), pp. 509–512.

[160] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt. "The magic of logical inference in probabilistic programming". In: *Theory and Practice of Logic Programming* 11.4-5 (2011), pp. 663–680.

[161] D. Nitti, T. De Laet, and L. De Raedt. "Probabilistic logic programming for hybrid relational domains". In: *Machine Learning* 103.3 (2016), pp. 407–449. ISSN: 1573-0565. DOI: 10.1007/s10994-016-5558-8.

[162] R. M. Fung and K.-C. Chang. "Weighing and Integrating Evidence for Stochastic Simulation in Bayesian Networks". In: *Fifth Annual Conference on Uncertainty in Artificial Intelligence*. North-Holland Publishing Co. 1990, pp. 209–220.

[163] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. Cambridge, MA: MIT Press, 2009. ISBN: 9780262013192.

[164] F. Wood, J. W. van de Meent, and V. Mansinghka. "A New Approach to Probabilistic Programming Inference". In: *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 2014, pp. 1024–1032.

[165] T. Lager and J. Wielemaker. "Pengines: Web Logic Programming Made Easy". In: *Theory and Practice of Logic Programming* 14.4-5 (2014), pp. 539–552. DOI: 10.1017/S1471068414000192. URL: http://dx.doi.org/10.1017/S1471068414000192.

[166] J. Wielemaker, Z. Huang, and L. van der Meij. "SWI-Prolog and the web". In: *Theory and Practice of Logic Programming* 8.3 (2008), pp. 363–392. DOI: 10.1017/S1471068407003237.

[167] A. Nguembang Fadja and F. Riguzzi. "Probabilistic Logic Programming in Action". In: *Towards Integrative Machine Learning and Knowledge Extraction*. Ed. by A. Holzinger, R. Goebel, M. Ferri, and V. Palade. Vol. 10344. Springer. Heidelberg, Germany: Springer, 2017. DOI: 10.1007/978-3-319-69775-8_5.

[168] A. Hyttinen, F. Eberhardt, and M. Järvisalo. "Do-calculus when the True Graph Is Unknown." In: *31st International Conference on Uncertainty in Artificial Intelligence (UAI-15)*. 2015, pp. 395–404.

[169] M. A. Islam, C. Ramakrishnan, and I. Ramakrishnan. "Inference in probabilistic logic programs with continuous random variables". In: *Theory and Practice of Logic Programming* 12 (Special Issue 4-5 2012), pp. 505–523. ISSN: 1475-3081.

[170] L. Bühmann, J. Lehmann, and P. Westpha. "DL-Learner – A framework for inductive learning on the Semantic Web". In: *Journal of Web Semantics* 39 (2016), pp. 15–24.

[171] V. Vassiliadis, J. Wielemaker, and C. Mungall. "Processing OWL2 Ontologies using Thea: An Application of Logic Programming". In: *Proceedings of the 6th International Workshop on OWL: Experiences and Directions*. Vol. 529. CEUR Workshop Proceedings. CEUR-WS.org, 2009.

[172] M. Codish, V. Lagoon, and P. J. Stuckey. "Logic programming with satisfiability". In: *Theory and Practice of Logic Programming* 8.1 (2008), pp. 121–128.

[173] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by E. Giunchiglia and A. Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518.

[174] R. Penaloza and B. Sertkaya. "Axiom Pinpointing is Hard." In: *Description Logics* 477 (2009).

[175] R. Penaloza and B. Sertkaya. "Complexity of axiom pinpointing in the DL-Lite family". In: *23rd International Workshop on Description Logics*. 2010, p. 173.

[176] R. Penaloza and B. Sertkaya. "Complexity of Axiom Pinpointing in the DL-Lite Family of Description Logics." In: *Proceedings of the 19th European Conference on Artificial Intelligence*. Vol. 215. 2010, pp. 29–34.

[177] F. Baader, R. Penaloza, and B. Suntisrivaraporn. "Pinpointing in the Description Logic $\mathcal{EL}^+$". In: *Applied Artificial Intelligence* (2007), p. 52.

[178] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. "On Generating All Maximal Independent Sets". In: *Information Processing Letters* 27.3 (1988), pp. 119–123.

[179] A. Rauzy, E. Châtelet, Y. Dutuit, and C. Bérenguer. "A practical comparison of methods to assess sum-of-products". In: *Reliability Engineering and System Safety* 79.1 (2003), pp. 33–42.

[180] F. Riguzzi. "A Top Down Interpreter for LPAD and CP-logic". In: *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence, Rome*. Vol. 4733. LNAI. Springer, 2007, pp. 109–120. DOI: 10.1007/978-3-540-74782-6_11.

[181] F. Riguzzi and T. Swift. "Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions". In: *26th International Conference on Logic Programming*. Vol. 7. LIPIcs. Saarbrücken, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 162–171. ISBN: 978-3-939897-17-0. DOI: 10.4230/LIPIcs.ICLP.2010.162.

[182] F. Riguzzi and T. Swift. "Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics". In: *Theory and Practice of Logic Programming* 13.2 (2013), pp. 279–302. DOI: 10.1017/S1471068411000664.

[183] F. Riguzzi. "Speeding Up Inference for Probabilistic Logic Programs". In: *The Computer Journal* 57.3 (2014), pp. 347–363. DOI: 10.1093/comjnl/bxt096.

[184] M. Chavira and A. Darwiche. "On probabilistic inference by weighted model counting". In: *Artif. Intell.* 172.6-7 (2008), pp. 772–799.

[185] F. Riguzzi, E. Lamma, E. Bellodi, and R. Zese. "BUNDLE: A Reasoner for Probabilistic Ontologies". In: *7th International Conference on Web Reasoning and Rule Systems (RR 2013), Mannheim, Germany*. Ed. by W. Faber and D. Lembo. Vol. 7994. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 183–197. ISBN: 978-3-642-39665-6. DOI: 10.1007/978-3-642-39666-3_14.

[186] P. Klinov and B. Parsia. "Optimization and Evaluation of Reasoning in Probabilistic Description Logic: Towards a Systematic Approach". In: *The Semantic Web - ISWC 2008 - 7th International Semantic Web Conference*. Ed. by A. P. Sheth et al. Vol. 5318. Lecture Notes in Computer Science. Springer, 2008, pp. 213–228.

[187] E. Demir et al. "The BioPAX community standard for pathway data sharing". In: *Nature biotechnology* 28.9 (2010), pp. 935–942.

[188] G. Nagypál, R. Deswarte, and J. Oosthoek. "Applying the Semantic Web: The VICODI Experience in Creating Visual Contextualization for History". In: *Literary and Linguistic Computing* 20.3 (2005), pp. 327–349.

[189] B. Beckert and J. Posegga. "leanTAP: Lean Tableau-based Deduction". In: *Journal of Automated Reasoning* 15.3 (1995), pp. 339–358.

[190] A. Meissner. "An automated deduction system for description logic with ALCN language". In: *Studia z Automatyki i Informatyki* 28-29 (2004), pp. 91–110.

[191] T. Herchenröder. "Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics". MA thesis. School of Informatics, University of Edinburgh, 2006.

[192] I. Faizi. *A Description Logic Prover in Prolog, Bachelor's thesis, Informatics Mathematical Modelling, Technical University of Denmark*. 2011.

[193] G. Lukácsy and P. Szeredi. "Efficient description logic reasoning in Prolog: The DLog system". In: *Theory and Practice of Logic Programming* 9.3 (2009), pp. 343–414.

[194] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, and N. Leone. "OntoDLV: An ASP-based System for Enterprise Ontologies". In: *Journal of Logic and Computation* 19.4 (2009), pp. 643–670.

[195] P. Klinov. "Pronto: A Non-monotonic Probabilistic Description Logic Reasoner". In: *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*. Ed. by S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis. Vol. 5021. Lecture Notes in Computer Science. Springer, 2008, pp. 822–826.

[196] M. Bruynooghe et al. "ProbLog Technology for Inference in a Probabilistic First Order Logic". In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 719–724.

[197] N. J. Nilsson. "Probabilistic Logic". In: *Artificial Intelligence* 28.1 (1986), pp. 71–87.

[198] İ. İ. Ceylan and R. Peñaloza. "Bayesian Description Logics". In: *Informal Proceedings of the 27th International Workshop on Description Logics, Vienna, Austria, July 17-20, 2014*. Ed. by M. Bienvenu, M. Ortiz, R. Rosati, and M. Simkus. Vol. 1193. CEUR Workshop Proceedings. Aachen: Sun SITE Central Europe, 2014, pp. 447–458.

[199] İ. İ. Ceylan and R. Peñaloza. "Probabilistic Query Answering in the Bayesian Description Logic *BE*L". In: *Proceedings of Scalable Uncertainty Management - 9th International Conference, SUM 2015*. Ed. by C. Beierle and A. Dekhtyar. Vol. 9310. Lecture Notes in Computer Science. Springer, 2015, pp. 21–35.

[200] İ. İ. Ceylan, J. Mendez, and R. Peñaloza. "The Bayesian Ontology Reasoner is BORN!" In: *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description Logics (DL 2015)*. Ed. by M. Dumontier et al. Vol. 1387. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 8–14.

[201] M. Gavanelli, E. Lamma, F. Riguzzi, E. Bellodi, R. Zese, and G. Cota. "Abductive Logic Programming for Datalog± Ontologies". In: *Proceedings of the 30th Italian Conference on Computational Logic (CILC2015), Genova, Italy, 1-3 July 2015*. Ed. by D. Ancona, M. Maratea, and V. Mascardi. CEUR Workshop Proceedings 1459. Aachen, Germany: Sun SITE Central Europe, 2015, pp. 128–143. URL: http://ceur-ws.org/Vol-1459/paper21.pdf.

[202] M. Gavanelli, E. Lamma, F. Riguzzi, E. Bellodi, R. Zese, and G. Cota. "An Abductive Framework for Datalog± Ontologies". In: *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*. Ed. by M. D. Vos, T. Eiter, Y. Lierler, and F. Toni. Vol. 1433. CEUR Workshop Proceedings. CEUR-WS.org, 2015.

[203] S. Muggleton. "Inductive logic programming". In: *New generation computing* 8.4 (1991), pp. 295–318.

[204] S. Muggleton and L. De Raedt. "Inductive logic programming: Theory and methods". In: *Journal of Logic Programming* 19 (1994), pp. 629–679.

[205] L. De Raedt and K. Kersting. "Probabilistic Inductive Logic Programming". In: *Algorithmic Learning Theory, 15th International Conference, ALT 2004, Padova, Italy, October 2-5, 2004, Proceedings*. Ed. by S. Ben-David, J. Case, and A. Maruoka. Vol. 3244. Lecture Notes in Computer Science. Springer, 2004, pp. 19–36. DOI: 10.1007/978-3-540-30215-5_3.

[206] L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, eds. *Probabilistic Inductive Logic Programming*. Vol. 4911. LNCS. Springer, 2008. ISBN: 978-3-540-78651-1.

[207] S. Muggleton et al. "Stochastic logic programs". In: *Advances in inductive logic programming* 32 (1996), pp. 254–264.

[208] S. Muggleton. "Learning Stochastic Logic Programs". In: *Electron. Trans. Artif. Intell.* 4.B (2000), pp. 141–153.

[209] L. De Raedt and S. Dzeroski. "First-Order jk-Clausal Theories are PAC-Learnable". In: *Artificial Intelligence* 70.1-2 (1994), pp. 375–392.

[210] N. Helft. "Induction as Nonmonotonic Inference." In: *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*. 1989, pp. 149–156.

[211] E. Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

[212] S.-H. Nienhuys-Cheng and R. de Wolf, eds. *Foundations of Inductive Logic Programming*. Vol. 1228. LNCS. Springer, 1997. ISBN: 3-540-62927-0.

[213] L. De Raedt. *Logical and Relational Learning*. Cognitive Technologies. Springer, 2008. ISBN: 978-3-540-20040-6. DOI: 10.1007/978-3-540-68856-3. URL: http://dx.doi.org/10.1007/978-3-540-68856-3.

[214] L. D. Raedt, K. Kersting, S. Natarajan, and D. Poole. "Statistical relational artificial intelligence: Logic, probability, and computation". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 10.2 (2016), pp. 1–189.

[215] L. De Raedt and K. Kersting. "Probabilistic logic learning". In: *ACM SIGKDD Explorations Newsletter* 5.1 (2003), pp. 31–48.

[216] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal Statistical Society. Series B (methodological)* (1977), pp. 1–38.

[217] T. Sato and Y. Kameya. "Parameter Learning of Logic Programs for Symbolic-Statistical Modeling". In: *Journal of Artificial Intelligence Research* 15 (2001), pp. 391–454.

[218] F. Riguzzi and N. Di Mauro. "Applying the Information Bottleneck to Statistical Relational Learning". In: *Machine Learning* 86.1 (2012), pp. 89–114. DOI: 10.1007/s10994-011-5247-6.

[219] B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. "Parameter Learning in Probabilistic Databases: A Least Squares Approach". In: *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008)*. Vol. 5211. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 473–488.

[220] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. "Propositionalizing the EM algorithm by BDDs". In: *Late Breaking Papers of the 18th International Conference on Inductive Logic Programming (ILP 2008)*. 2008, pp. 44–49.

[221] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[222] S. Muggleton. "Inverse Entailment and Progol". In: *New Generation Computing* 13 (1995), pp. 245–286.

[223] C. Chu et al. "Map-Reduce for Machine Learning on Multicore". In: *Advances in Neural Information Processing Systems 19 (NIPS 2006)*. Ed. by B. Schölkopf, J. C. Platt, and T. Hoffman. MIT Press, 2006, pp. 281–288. ISBN: 0-262-19568-2. URL: http://papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.

[224] H. Khosravi, O. Schulte, J. Hu, and T. Gao. "Learning compact Markov logic Networks with decision trees". In: *Machine Learning* 89.3 (2012), pp. 257–277.

[225] A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. "Theories for Mutagenicity: A Study in First-Order and Feature-Based Induction". In: *Artificial Intelligence* 85.1-2 (1996), pp. 277–299.

[226] S. Kok and P. Domingos. "Learning the structure of Markov Logic Networks". In: *22nd international conference on Machine learning*. ACM, 2005, pp. 441–448.

[227] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. "Carcinogenesis Predictions Using ILP". In: *7th International Workshop on Inductive Logic Programming*. Ed. by N. Lavrac and S. Dzeroski. Vol. 1297. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 273–287.

[228] L. Mihalkova and R. J. Mooney. "Bottom-up Learning of Markov Logic Network Structure". In: *24th International Conference on Machine Learning*. ACM, 2007, pp. 625–632.

[229] N. Beerenwinkel et al. "Learning Multiple Evolutionary Pathways from Cross-Sectional Data". In: *Journal of Computational Biology* 12 (2005), pp. 584–598.

[230] M. Craven and S. Slattery. "Relational Learning with Statistical Predicate Invention: Better Models for Hypertext". In: *Machine Learning* 43.1-2 (2001), pp. 97–119.

[231] N. Di Mauro, E. Bellodi, and F. Riguzzi. "Bandit-based Monte-Carlo structure learning of probabilistic logic programs". In: *Machine Learning* 100.1 (2015), pp. 127–156. DOI: 10.1007/s10994-015-5510-3.

[232] E. Bellodi and F. Riguzzi. "Learning the Structure of Probabilistic Logic Programs". In: *ILP 2012*. Ed. by S. Muggleton, A. Tamaddoni-Nezhad, and F. Lisi. Vol. 7207. LNCS. Springer Berlin Heidelberg, 2012, pp. 61–75.

[233] T. N. Huynh and R. J. Mooney. "Discriminative structure and parameter learning for Markov logic networks". In: *25th international conference on Machine learning*. Ed. by W. W. Cohen, A. McCallum, and S. T. Roweis. ACM, 2008, pp. 416–423. ISBN: 978-1-60558-205-4.

[234] T. Khot, S. Natarajan, K. Kersting, and J. W. Shavlik. "Learning Markov Logic Networks via Functional Gradient Boosting." In: *Proceedings of the 11th IEEE International Conference on Data Mining*. IEEE, 2011, pp. 320–329.

[235] S. Natarajan, T. Khot, K. Kersting, B. Gutmann, and J. Shavlik. "Gradient-based Boosting for Statistical Relational Learning: The Relational Dependency Network Case". In: *Machine Learning* 86.1 (2012), pp. 25–56.

[236] E. Bellodi and F. Riguzzi. "Experimentation of an Expectation Maximization Algorithm for Probabilistic Logic Programs". In: *Intelligenza Artificiale* 8.1 (2012), pp. 3–18. DOI: 10.3233/IA-2012-0027.

[237] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. *Propositionalizing the EM algorithm by BDDs*. Tech. rep. TR08-0004. Dep. of Computer Science, Tokyo Institute of Technology, 2008.

[238] J. Lehmann et al. "DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web – Interoperability, Usability, Applicability* 6.2 (2015), pp. 167–195.

[239] J. Lehmann, S. Auer, L. Bühmann, and S. Tramp. "Class expression learning for ontology engineering". In: *Journal of Web Semantics* 9.1 (2011), pp. 71–81.

[240] J. Lehmann and P. Hitzler. "Concept learning in description logics using refinement operators". In: *Machine Learning* 78.1-2 (2010), p. 203.

[241] J. Lehmann and P. Hitzler. "A refinement operator based learning algorithm for the ALC description logic". In: *17th International Conference on Inductive Logic Programming*. Vol. 4894. Springer, 2007, pp. 147–160.

[242] J. Lehmann and P. Hitzler. "Foundations of refinement operators for description logics". In: *17th International Conference on Inductive Logic Programming*. Vol. 4894. Springer, 2007, pp. 161–174.

[243] A. Agresti and B. A. Coull. "Approximate is better than "exact" for interval estimation of binomial proportions". In: *The American Statistician* 52.2 (1998), pp. 119–126.

[244] S. Hellmann, J. Lehmann, and S. Auer. "Learning of OWL Class Descriptions on Very Large Knowledge Bases". In: *International Journal on Semantic Web and Information Systems* 5.2 (2009), pp. 25–48.

[245] J. Völker and M. Niepert. "Statistical schema induction". In: *The Semantic Web: Research and Applications*. Springer Berlin Heidelberg, 2011, pp. 124–138.

[246] D. Fleischhacker and J. Völker. "Inductive learning of disjointness axioms". In: *On the Move to Meaningful Internet Systems: OTM 2011*. Springer, 2011, pp. 680–697.

[247] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules in Large Databases." In: *International Conference on Very Large Data Bases*. Morgan Kaufmann, 1994, pp. 487–499.

[248] P. Minervini, C. d'Amato, and N. Fanizzi. "Learning probabilistic description logic concepts: Under different assumptions on missing knowledge". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 378–383.

[249] T. Riechert, K. Lauenroth, J. Lehmann, and S. Auer. "Towards semantic based requirements engineering". In: *Proceedings of the 7th International Conference on Knowledge Management*. 2007, pp. 144–151.

[250] T. Tudorache, J. Vendetti, and N. F. Noy. "Web-Protege: A Lightweight OWL Ontology Editor for the Web." In: *OWLED*. Vol. 432. 2008.

[251] S. Hellmann, J. Lehmann, and S. Auer. "Learning of OWL Class Expressions on Very Large Knowledge Bases and its Applications". In: 2011. Chap. 5, pp. 104–130.

[252] G. T. Williams, J. Weaver, M. Atre, and J. A. Hendler. "Scalable reduction of large datasets to interesting subsets". In: *Journal of Web Semantics* 8.4 (2010), pp. 365–373.

[253] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. "LOD laundromat: a uniform way of publishing other people's dirty data". In: *13th International Semantic Web Conference*. Springer. 2014, pp. 213–228.

[254] F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. "Structure Learning of Probabilistic Logic Programs by MapReduce". In: *25th International Conference on Inductive Logic Programming*. Ed. by K. Inoue, H. Ohwada, and A. Yamamoto. 2015.

[255] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. "Description Logic Programs: Combining Logic Programs with Description Logic". In: *Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003*. Ed. by G. Hencsey, B. White, Y. R. Chen, L. Kovács, and S. Lawrence. New York, NY, USA: ACM, 2003, pp. 48–57. DOI: 10.1145/775152.775160. URL: http://doi.acm.org/10.1145/775152.775160.

[256] A. d. Garcez, K. Broda, and D. M. Gabbay. "Symbolic knowledge extraction from trained neural networks: A sound approach". In: *Artificial Intelligence* 125.1 (2001), pp. 155–207.

[257] A. S. A. Garcez and G. Zaverucha. "The connectionist inductive learning and logic programming system". In: *Applied Intelligence* 11.1 (1999), pp. 59–77.

[258] J. Lehmann, S. Bader, and P. Hitzler. "Extracting reduced logic programs from artificial neural networks". In: *Applied Intelligence* 32.3 (2010), pp. 249–266.

[259] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. "A review of relational machine learning for knowledge graphs". In: *Proceedings of the IEEE* 104.1 (2016), pp. 11–33.

Appendix A

List of Publications

The work described in this thesis is based on the following publications.

Journal papers

- *Fabrizio Riguzzi, Giuseppe Cota, Elena Bellodi, and Riccardo Zese*
  Causal inference in cplint
  In: International Journal of Approximate Reasoning 91 (2017), pp. 216–232. doi: 10.1016/j.ijar.2017.09.007.

- *Marco Alberti, Elena Bellodi, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese*
  cplint on SWISH: Probabilistic Logical Inference with a Web Browser
  In: Intelligenza Artificiale 11 (2017), pp. 47–64. doi: 10.3233/IA-170105.

- *Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Riccardo Zese, and Giuseppe Cota*
  A web system for reasoning with probabilistic OWL
  In: Software, Practice and Experience 47 (2017), pp. 125–142. doi: 10.1002/spe.2410.

- *Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma*
  A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics
  In: International Journal of Approximate Reasoning 80 (2017), pp. 313–333. doi: 10.1016/j.ijar.2016.10.002.

- *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota*
  Probabilistic logic programming on the web
  In: Software, Practice and Experience 46 (2016), pp. 1381–1396. doi: 10.1002/spe.2386.

- *Riccardo Zese, Elena Bellodi, Fabrizio Riguzzi, Giuseppe Cota, and Evelina Lamma*
  Tableau reasoning for description logics and its extension to probabilities
  In: Annals of Mathematics and of Artificial Intelligence (2016), pp. 1–30. doi: 10.1007/s10472-016-9529-3.

Book chapters

- *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota*
  Learning Probabilistic Description Logics
  In: Uncertainty Reasoning for the Semantic Web III. ISWC International Workshops, URSW 2011-2013, Revised Selected Papers. Vol. 8816. Berlin: Springer Verlag, 2014, pp. 63–78. doi: 10.1007/978-3-319-13413-0_4.

Conference and workshop papers

- *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
  Abductive Logic Programming for Normative Reasoning and Ontologies
  In: New Frontiers in Artificial Intelligence (JSAI-isAI 2015 Workshops, LENLS, JURISIN, AAA, HAT-MASH, TSDAA, ASD-HR and SKL, Kanagawa, Japan, November 16-18, 2015, Revised Selected Papers). Vol. 10091. Springer, 2017, pp. 187–203. doi: 10.1007/978-3-319-50953-2_14.

- *Fabrizio Riguzzi, Riccardo Zese, and Giuseppe Cota*
  Probabilistic inductive logic programming on the web
  In: 20th International Conference on Knowledge Engineering and Knowledge Management, EKAW 2016. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 10180. Springer Verlag, 2017, pp. 172–175. doi: 10.1007/978-3-319-58694-6_25.

- *Giuseppe Cota*
  Systems and Learning Algorithms for Probabilistic Logical Knowledge Bases
  In: Proceedings of the Doctoral Consortium of AI*IA 2016 co-located with the 15th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2016). Vol. 1769. Aachen: Sun SITE Central Europe, 2017, pp. 5–10.

- *Fabrizio Riguzzi, Evelina Lamma, Marco Alberti, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
  Probabilistic logic programming for natural language processing
  In: URANIA 2016, Deep Understanding and Reasoning: A Challenge for Next-generation Intelligent Agents, Proceedings of the AI*IA Workshop on Deep Understanding and Reasoning: A Challenge for Next-generation Intelligent Agents 2016. Vol. 1802. Aachen: Sun SITE Central Europe, 2017, pp. 30–37.

- *Fabrizio Riguzzi, Riccardo Zese, and Giuseppe Cota*
  Probabilistic inductive logic programming on the web
  In: 20th International Conference on Knowledge Engineering and Knowledge

Management, EKAW 2016. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 10180. Springer Verlag, 2017, pp. 172–175. doi: 10.1007/978-3-319-58694-6_25.

- *Marco Alberti, Elena Bellodi, Giuseppe Cota, Evelina Lamma, Fabrizio Riguzzi, and Riccardo Zese*
  Probabilistic Constraint Logic Theories
  In: Proceedings of the 3nd International Workshop on Probabilistic Logic Programming (PLP). Vol. 1661. Aachen: Sun SITE Central Europe, 2016, pp. 15–28.

- *Marco Alberti, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese*
  Probabilistic Logical Inference On the Web
  In: Proceedings of the 15th Conference of the Italian Association for Artificial Intelligence (AI*IA2016), Genova, Italy, 28 November - 1 December 2016. Vol. 10037. Heidelberg: Springer International Publishing, 2016, pp. 351–363. doi: 10.1007/978-3-319-49130-1_26.

- *Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma*
  Scaling Structure Learning of Probabilistic Logic Programs by MapReduce
  In: 22nd European Conference on Artificial Intelligence (ECAI 2016). Vol. 285. Amsterdam: IOS Press, 2016, pp. 1602–1603. doi: 10.3233/978-1-61499-672-9-1602.

- *Giuseppe Cota, Riccardo Zese, Elena Bellodi, Fabrizio Riguzzi, and Evelina Lamma*
  Distributed Parameter Learning for Probabilistic Ontologies
  In: Inductive Logic Programming: 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers. Vol. 9575. Heidelberg: Springer International Publishing Switzerland, 2016, pp. 30–45. doi: 10.1007/978-3-319-40566-7_3.

- *Giuseppe Cota, Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
  Learning Probabilistic Ontologies with Distributed Parameter Learning In: Proceedings of the Doctoral Consortium (DC) co-located with the 14th Conference of the Italian Association for Artificial Intelligence (AI*IA 2015). Vol. 1485. Aachen: Sun SITE Central Europe, 2015, pp. 7–12.

- *Giuseppe Cota, Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
  Structure Learning with Distributed Parameter Learning for Probabilistic Ontologies
  In: Proceedings of the ECMLPKDD 2015 Doctoral Consortium. Aalto: Aalto University Library, 2015, pp. 75–84.

- *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
  Abductive logic programming for Datalog± ontologies
  In: CILC 2015 Italian Conference on Computational Logic. Proceedings of the

30th Italian Conference on Computational Logic. Vol. 1459. CEUR Workshop proceedings, 2015, pp. 128–143.

- *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
  An Abductive Framework for Datalog± Ontologies
  In: Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015). Vol. 1433. CEUR Workshop proceedings, 2015, pp. 1–13.

Miscellaneous

- *Fabrizio Riguzzi and Giuseppe Cota*
  Probabilistic Logic Programming Tutorial
  The Association for Logic Programming Newsletter. 2016. url: `http://www.cs.nmsu.edu/ALP/2016/03/probabilistic-logic-programming-tutorial/`.

To be published

- *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
  Reasoning on Datalog± Ontologies with Abductive Logic Programming
  In: Fundamenta Informaticae, Special Issue CILC 2015.