UNIVERSITÀ DEGLI STUDI DI FERRARA

FACOLTÀ DI INGEGNERIA

DOTTORATO DI RICERCA IN SCIENZE DELL'INGEGNERIA

Ciclo XXVIII

COORDINATORE Prof. Stefano Trillo

SETTORE SCIENTIFICO DISCIPLINARE ING-INF/05

Probabilistic Reasoning and Learning for the Semantic Web

Dottorando

Dott. Zese Riccardo

Tutore

Prof. Riguzzi Fabrizio

Correlatore

Prof.ssa Lamma Evelina

Anni 2013/2015

Abstract

The Semantic Web introduced a new vision of the World Wide Web where the information resources published on the Internet are readable and understandable by machines. However, incompleteness and/or uncertainty are intrinsic to much information, specially when it is collected from different sources. Thus we need a way to manage this kind of data.

In this thesis we address this problem and we present a complete framework for handling uncertainty in the Semantic Web. Description Logics (DLs) are the basis of the Semantic Web. DL knowledge bases (KBs) contains both assertional and terminological information regarding individuals, classes of individuals and relationships among them. We first defined a probabilistic semantics for DLs, called DISPONTE. It is inspired by the distribution semantics, a well known approach in probabilistic logic programming. DISPONTE permits to associate degrees of belief to pieces of information and to compute the probability of queries to KBs.

The thesis then proposes a suite of algorithms for reasoning with KBs following DISPONTE:

- BUNDLE, for "Binary decision diagrams for Uncertain reasoNing on Description Logic thEories", computes the probability of queries w.r.t. DISPONTE KBs by means of the tableau algorithm and knowledge compilation. BUNDLE is based on Pellet, a state of the art reasoner, and is written in Java.

- TRILL, for "Tableau Reasoner for descrIption Logics in Prolog", performs inference over DISPONTE KBs with the tableau algorithm implemented in the declarative Prolog language. Prolog is useful for managing the non-determinism of the reasoning process.

- TRILL^P, for "TRILL powered by Pinpointing formulas", differs from TRILL because it encodes the set of all explanations for queries with a more compact Boolean formula.

A second problem to address is the fact that the probability values are difficult to set for humans. However, usually information is available which can be leveraged for tuning these parameters. Moreover, terminological information in KBs may be incomplete or poorly structured. We thus need of learning systems able to cope with these problems. We present two learning systems, one for each problem:

- EDGE, for "Em over bDds for description loGics paramEter learning", learns the parameters of a DISPONTE KB.
- LEAP, for "LEArning Probabilistic description logics", learns terminological axioms together with their parameters by using EDGE.

However, the size of the data is constantly increasing, leading to the so-called Bid Data, Dataset are often too huge to be handled by a single machine in a reasonable time. Modern computing infrastructures such as clusters and clouds must be used where the work is divided among different machines. We thus extended both EDGE and LEAP to exploit these facilities by implementing EDGE$^{\mathrm{MR}}$ and LEAP$^{\mathrm{MR}}$ that distribute the work using a MapReduce approach.

All systems were tested on real life problems and their performances was comparable or superior to the state of the art.

Sinossi

Il Semantic Web è basato su una nuova visione del World Wide Web in cui le informazioni contenute nelle risorse pubblicate sono leggibili e gestibili dalle macchine. Tuttavia, queste informazioni sono spesso incomplete e/o incerte, specialmente quando vengono raccolte da diverse sorgenti. Risulta quindi necessario gestirle in maniera appropriata.

In questa tesi ci siamo concentrati su questo problema, presentando un insieme completo di strumenti per gestire l'incertezza nel contesto del Semantic Web. Le logiche descrittive (LD) rappresentano la base del Semantic Web. Le basi di conoscenza espresse con le LD contengono informazioni sia asserzionali sia terminologiche riguardanti individui, classi di individui e le relazioni che intercorrono fra loro. Il primo passo è stato la definizione di una semantica probabilistica per LD, chiamata DISPONTE. Essa è ispirata alla semantica distributiva, molto diffusa nel campo della programmazione logico-probabilistica. DISPONTE permette di associare gradi di fiducia a porzioni di informazione e di calcolare la probabilità delle interrogazioni basandosi su basi di conoscenza probabilistiche.

La tesi propone inoltre un insieme di algoritmi capaci di ragionare su basi di conoscenza che seguono DISPONTE:

- BUNDLE, acronimo di "Binary decision diagrams for Uncertain reasoNing on Description Logic thEories", calcola la probabilità di interrogazioni rispetto ad una base di conoscenza DISPONTE sfruttando l'algoritmo tableau e tecniche di knowledge compilation. BUNDLE è basato sul noto ragionatore Pellet ed è interamente scritto in Java.

- TRILL, acronimo di "Tableau Reasoner for descrIption Logics in Prolog", esegue inferenza su basi di conoscenza DISPONTE sfruttando un'implementazione dell'algoritmo tableau scritta in Prolog, utile per gestire il non-determinismo intrinseco nel processo di inferenza.

- TRILL^P, acronimo di "TRILL powered by Pinpointing formulas", differisce da TRILL nella codifica dell'insieme di spiegazioni che risulta essere, in questo secondo algoritmo, più compatta.

Un secondo problema risiede nel fatto che i valori di probabilità sono difficili da definire per gli esseri umani. Normalmente però si hanno a disposizione informazioni sul dominio che possono essere sfruttate per definire questi parametri. Inoltre, le informazioni terminologiche contenute nelle basi di conoscenza sono spesso incomplete e scarsamente strutturate. In questa tesi vengono presentati due sistemi di apprendimento che risolvono i problemi sopra citati:

- EDGE, acronimo di "Em over bDds for description loGics paramEter learning", apprende i parametri di una base di conoscenza DISPONTE.
- LEAP, acronimo di "LEArning Probabilistic description logics", apprende assiomi terminologici insieme ai parametri associati usando EDGE.

Va inoltre notato che negli ultimi anni la quantità di dati da gestire sta costantemente e rapidamente crescendo, portando alla nascita del concetto di Big Data. La quantità di dati risulta troppo grande per poter essere gestita da una singola macchina in tempi ragionevoli. Le moderne infrastrutture di calcolo come i cluster e il cloud devono essere sfruttati per poter dividere il carico di lavoro su più nodi. Abbiamo quindi esteso EDGE e LEAP per utilizzare queste infrastrutture implementando EDGE$^{\mathrm{MR}}$ e LEAP$^{\mathrm{MR}}$ che impiegano un approccio MapReduce per distribuire il lavoro.

Tutti i sistemi sono stati testati su problemi reali e le loro prestazioni sono risultate comparabili o superiori agli approcci considerati lo stato dell'arte.

Acknowledgements

My heartfelt thanks to Prof. Fabrizio Riguzzi and Prof. Evelina Lamma to whom goes my deepest gratitude for the invaluable help provided, their constant availability and the interest and the passion that they instilled in me for these topics. Their suggestions guided me throughout the time of research and writing of this thesis.

I thank my lab fellows for the stimulating discussions, for all the hours spent working together and for all the fun we have had in these years.

A very special thanks goes to my family, to whom this dissertation is dedicated. I cannot thank my parents and my sister enough for the love they gave me, none of this would have been possible without their constant support and encouragement.

Thanks to all my friends, a second family for me, for every wonderful moment lived together.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction

Chapter 1

Semantic Web

The term Semantic Web was coined by Tim Berners Lee and specifies an evolution of the World Wide Web in which the published documents (HTML pages, files, etc.) are associated with metadata. Metadata are information which specifies the semantic context following a format suitable for automatic querying, elaboration and interpretation. Hence, the focus has moved to the description of the meaning of information contained in the resources and how these resources are related to each other. In this way, the Web may be processed independently by machines, without help from human users.

The evolution of the Web in the Semantic direction started in 1999 when the Word Wide Web Consortium (W3C) defined the standards Resource Description Framework (RDF) and RDF Schema (RDFS). Both RDF and RDFS are based on XML and prescribe a way to define relationships between information by taking inspiration from predicate logic (or first order logic) and by exploiting typical Web's and XML's standards such as URI and namespaces. RDF and RDFS are languages that have a limited expressiveness. RDF is based on triples `<subject, predicate, object>` or `<resource, attribute, value>`, therefore it allows only the definition of binary predicates over terms of the domain of discourse. For example, if want to state that "Yoda is a Jedi Master", the corresponding RDF representation is `<Yoda, type, Jedi Master>`. In particular, in this example "Jedi Master" is a class that represents a set of objects of the domain but it can be seen also as an object itself of the domain, while "type" is a built-in predicate of RDF indicating that an object belongs to a class. The corresponding binary predicate is *type*('*Yoda*','*Jedi Master*'). Do-

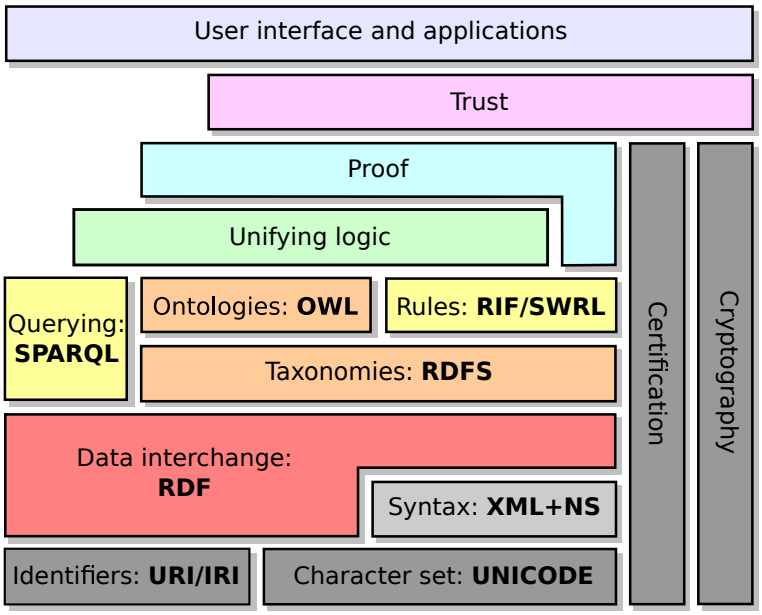


Figure 1.1: W3C Semantic Web Layer Cake.

main specific predicates, which link together two objects, can be defined too. They are also called properties. An example of a triple using a domain specific predicate is `<Rocky, calls, Adrian>`, where "Rocky" and "Adrian" are two individuals while the property "calls" means that an individual, in the example "Rocky", calls out repeatedly a second individual, "Adrian". In this case, the corresponding binary predicate is *calls*('*Rocky*','*Adrian*'). RDFS allows the definition of hierarchies between classes and properties and of constraints on the range and the domain of properties.

To overcome the limitations of RDF and RDFS, in 2004 the W3C formulated a new standard, richer and more expressive than RDF, called Web Ontology Language (OWL for short). OWL is part of a stack of W3C recommendations concerning the Semantic Web, shown in Figure 1.1.

The basis of the stack is formed by the naming mechanism (URI and IRI), the character encoding (Unicode) and the basic languages (such as XML and Namespaces), which specify the basic syntax. The third layer describes the information of the domain by means of RDF. Above, we can find query languages such as SPARQL, which allows querying data stored in different data sources modeled as RDF or viewed as RDF via middleware. SPARQL can extract data and perform queries that are typical of databases query languages.

Above SPARQL, RDF Schema defines RDF vocabularies. Further above we can find OWL together with languages for allowing extensions permitting the association of rules to data, i.e., *if-then-else* rules. The next layer offers reasoning systems to find new implicit information from the explicit one, also providing proofs for the inferred new knowledge. The layer above provide security mechanisms for encryption and data certification available across all the underlying layers. Finally, the topmost layer contains applications for final users.

1.1 Description Logics and Semantic Web

Formalisms for modeling information of a domain are used for formulating *ontologies* or *knowledge bases* (KBs), which group information from the domain of interest. Usually, KBs in the Semantic Web are formulated in OWL which is based on expressive Description Logics (DL). OWL was defined to provide a first-order formalism that is decidable and that allows associating a simple well-established declarative semantics to structured representations of knowledge.

Informally, a KB consists of a hierarchical description of concepts of the domain, together with the description of the properties of each concept and of the objects that belong to the concepts together with relations among them. Thus, KBs are crucial in the vision of the Semantic Web because they allow sharing terms among web resources in order to pave the way for the exploitation of (automatic) navigation, information discovery and retrieval and data integration.

KBs were used for modeling a wide variety of domains. Nowadays we can find standard KBs in fields such as chemistry, medicine, business, etc.

1.2 The Current Vision of the Semantic Web

Originally, each KB modeled a specific domain and was independent form the others. In the last few years, linking KBs, even from different areas of expertise, has become of foremost importance. This idea originated a huge net of information, called *Linked Open Data Cloud*, shown in Figure 1.2, in which each node is a KB and each link between nodes stands for all the relationships

between concepts from different KBs.

Recently, the definition of *Big Data* has appeared. Big Data indicates information whose size is too large for being processed and stored using standard approaches. The use of Big Data forces the adoption of distributed approaches using many workers, often located in different machines.

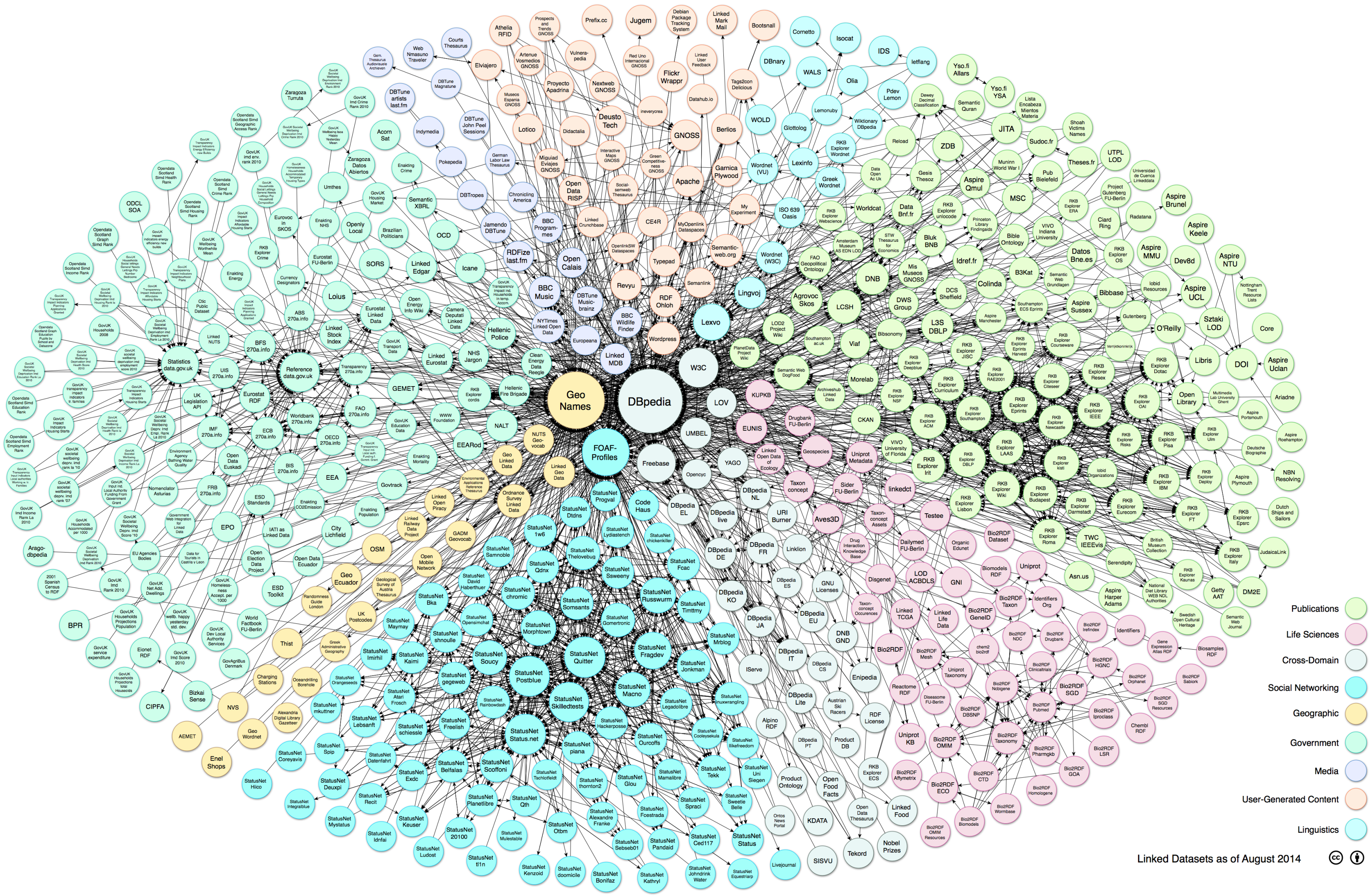


Figure 1.2: Open Linked Data Cloud. *Linking Open Data cloud diagram 2014, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak.* `http://lod-cloud.net/`

Chapter 2

Probability

The Semantic Web aims at modeling domains from the real worlds. In such scenarios much information has intrinsic uncertainty. This requires studying formalisms able to combine probability theory with logics and, in the case of the Semantic Web, with DLs.

Real data, such as images and structured and non-structured texts, comes from many different sources which are in most cases of varying trustworthiness, therefore information can be unsure or incorrect.

All these aspects require algorithms able not only to execute inference, but also to execute *probabilistic inference*, i.e., computing the probability of the truth of the inferred information, and to execute (*probabilistic*) *learning*. In this field learning takes two different connotations: (1) learning the parameters, i.e., the value of the probabilities associated with information and (2) learning new information or repairing already known information. The first is called *parameter learning*, while the second is named *structure learning*. Usually structure learning systems use parameter learning as a subroutine.

First it is of foremost importance to formally define probabilistic extensions for DLs which are applicable to OWL and, thus, to the Semantic Web.

In the past few decades, many probabilistic knowledge representation formalisms have been developed to cope with uncertainty. For example, in the last 20 years, significant advantages have been achieved in the field of Probabilistic Logic Programming (PLP) and many languages have been proposed that combine probability with logic programming. Since the beginning, together with the problem of performing inference, much attention was focused

on the problem of learning probabilistic logic programs. Learning these programs constitutes a whole subfield of Logic Programming, called Probabilistic Inductive Logic Programming (PILP).

Despite the large number of PLP proposals, the combination of probability theory and DLs is a young field. Given the connections between Logic Programming (LP) and DLs we believe that the results of PLP can be exploited for DLs. In this vain, we proposed DISPONTE.

DISPONTE applies the distribution semantics [129], one of the most common approaches in PLP, to DLs. The distribution semantics defines a probability distribution over normal logic programs, called worlds. Then, the distribution is extended to a joint distribution over worlds and a query from which the probability of a query can be obtained by marginalization.

In DISPONTE, each axiom can be annotated with a probability that specifies a degree of belief in the corresponding axiom. Therefore, a probabilistic DISPONTE KB defines a distribution over worlds obtained by including an axiom in a world with the probability given by the annotation.

The main issue is that performing inference by listing all the possible worlds is unfeasible since their number is exponential. Hence, it is necessary to study other techniques to compute the probability of queries.

2.1 Probabilistic Inference

In the last few years, many approaches have been presented to perform inference. One of the most used is the tableau algorithm. It starts from a graph that represents the set of individuals of the domain together with information of them. This graph is then expanded using a set of consistency preserving expansion rules. The tableau algorithm can be modified to return explanations for the query and knowledge compilation can be applied on it to compute the probability of queries from the explanations. Knowledge compilation transforms a Boolean formula into a circuit language such as Binary Decision Diagrams (BDDs). A BDD is a rooted graph where every node corresponds to a Boolean variable and has two children, one per Boolean value.

This approach has the problem that some of the tableau expansion rules are *non-deterministic*, so reasoners have to implement a search strategy in an *or-*

branching space to find all the possible explanations. These requires developing solutions to cope with backtracking because the algorithms have to explore all the non-deterministic choices done during the inference process. One possible solution is given by the Hitting Set algorithm [112] that repeatedly removes an axiom from the KB and then computes again a new explanation. Another solution consists in the use of declarative languages, such as Prolog, that have built-in backtracking facilities.

The possibility of using LP for implementing reasoners has been observed by many researchers. Some of them proposed, as we do, approaches exploiting LP-based implementations of the tableau [94, 59, 10]. Other proposals make use of inference methods based on standard LP resolution (e.g., in the DLog system by [86]), bottom-up inference methods, e.g., based on Answer Set Programming (e.g., ontoDLP, and its reasoner ontoDLV [115]), or the *chase* algorithm for Datalog$^{\pm}$ [19].

2.2 Probabilistic Learning

One of the main problems to solve in probabilistic knowledge representation is that defining the values of the probabilities contained in the KBs is difficult for humans. However, usually we have information on the modeled domain that can be leveraged to automatically tune these parameters. Moreover, information contained in KBs is often poorly structured or incomplete. There is thus the need to develop algorithms that can learn also the structure of KBs.

Finally, the widespread adoption of the Internet and the increasing number of resources available on the Web (also due to the so-called *Internet of Things*) provide the opportunity to gather huge sets of data. The ability to process and perform learning and inference over massive data is one of the major challenges of the current decade. Big data is strictly intertwined with the availability of scalable and distributed algorithms that exploit high performance computing infrastructures, such as clusters and clouds.

Chapter 3

Aims of the Thesis

The Semantic Web is constantly evolving and many efforts have been made to define standards for representing knowledge. What is missing in this context is a standard way to represent uncertain information. Many approaches have been presented, but most of them extend actual standards by introducing new formalisms.

This thesis aims at providing a complete framework for managing uncertainty in the Semantic Web, first by defining a probabilistic semantics and then by presenting algorithms able to handle it.

The proposed semantics, called DISPONTE, is based on the distribution semantics, presented in 1995 by Sato and applied by many Logic Programming languages. This approach is particularly appealing for its intuitiveness and because practical inference algorithms have been developed, which use Binary Decision Diagrams for the computation of the probability of queries.

The systems presented can be divided into inference and learning algorithms. For reasoning, three systems will be presented: (1) BUNDLE, a Java application, and (2) TRILL and (3) TRILL^P, written in Prolog in order to exploit Prolog management of non-determinism. For learning, we will present (1) EDGE, for learning the parameters of probabilistic KBs and (2) LEAP, for learning also the structure of probabilistic KBs. In this way we can repair incorrect and incomplete KBs and automatically set their parameters.

The goals of this thesis is on one hand to show that Prolog is a viable language to implement reasoning algorithms and on the other hand that, following the way paved by the diffusion of Big Data and Linked Open Data,

the need of distributed techniques are of foremost importance, leading to the implementation of parallelized and distributed inference and learning systems.

The effectiveness of the developed algorithms was tested on many tasks and on different datasets. All the algorithms show good performances, comparable or superior to the state of the art.

Chapter 4

Structure of the Thesis

The thesis is divided into six parts: Introduction, preliminaries of Description Logics, Probabilistic Description Logics, where a new semantics is proposed, Probabilistic Reasoning and Probabilistic Learning, where our algorithms are presented, Summary and Future works.

Part I starts with introductory chapters clarifying the nature, motivations, context and goals of this thesis. Chapter 5 lists the publications related to the thesis and the awards won.

Part II recalls basic concepts on knowledge bases and Description Logic (DL) languages. In particular, Chapter 6 describes the foundation of DLs which will be used throughout the thesis as the representation language. Chapter 7 discusses about the characteristics of DLs, their expressive powers and semantics. Chapter 8 shows significant examples of DLs, which will be further referred in the thesis. Chapter 9 defines the Web Ontology Language (OWL), which is the standard recommended by the W3C for the modelization of KBs. Chapter 10 presents the inference problem for DLs and possible solutions.

Part III discusses about probabilistic DLs and describes our probabilistic semantics DISPONTE. Chapter 11 illustrates the basis semantics of DISPONTE, the distribution demantics, also giving an introduction on Logic Programming, the field where this semantics born. Here, the problems of inference and of parameter and/or structure learning are discussed and the Prolog language is presented. This language will be used in the following for studying new approaches to inference for DLs. Chapter 12 formally defines the DISPONTE semantics, while Chapter 13 discusses probabilistic DLs alternative to DISPONTE.

Part IV covers probabilistic inference. Chapter 14 shows probabilistic inference techniques. Chapters 15, 16 and 17 present our inference algorithms: the first the system BUNDLE, the second the system TRILL and the last its evolution TRILL^P. Chapter 18 discusses about the complexity of the inference problem. Chapter 19 illustrates related work for (probabilistic) inference, while Chapter 20 describes the experiments made for BUNDLE, TRILL and TRILL^P.

Part V is dedicated to probabilistic learning problem. In particular, Chapter 22 presents EDGE by first describing the Expectation Maximization algorithm exploited by EDGE. Chapter 23 describes the system LEAP together with CELOE algorithm integrated in it. Chapter 24 discusses about the problem of distributing learning process and presents the evolutions of EDGE and LEAP to cope with parallelization, EDGE$^{\mathrm{MR}}$ and LEAP$^{\mathrm{MR}}$. Chapter 25 covers related learning systems and Chapter 26 illustrates the experiments made for our learning algorithms.

Part VI summarizes the research work conducted in this dissertation and presents directions for future work.

Chapter 5

Publications and Awards

Papers containing the work described in this thesis were presented in various venues:

- **Awards**
  - **RR 2013 Best Paper Award** for the paper:
    Fabrizio Riguzzi, Evelina Lamma, Elena Bellodi, and Riccardo Zese
    BUNDLE: A reasoner for probabilistic ontologies
    7th International Conference on Web Reasoning and Rule Systems, volume 7994 of Lecture Notes in Computer Science, pages 183-197, 2013.
  - **Premio per tesi di innovazione tecnologica in ambito open source** AA 2012/2013 for the master thesis:
    Probabilistic Reasoning on Ontologies, regarding first implementation of BUNDLE.
- **Journal papers**
  1. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota*
     Probabilistic Logic Programming on the Web
     Software: Practice and Experience, 2015. DOI: 10.1002/spe.2386
  2. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
     Probabilistic description logics under the distribution semantics

Semantic Web - Interoperability, Usability, Applicability, 6(5):447-501, 2015.

3. *Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese*
   A history of probabilistic inductive logic programming
   Frontiers in Robotics and AI, 1(6):1-5, 2014.
4. *Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Vitor Santos Costa, and Riccardo Zese*
   Lifted variable elimination for probabilistic logic programming
   Theory and Practice of Logic Programming, 14(Special issue 4-5 - ICLP 2014):681-695, 2014.

- **Book Chapters**

1. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota*
   Learning probabilistic description logics
   Uncertainty Reasoning for the Semantic Web III, volume 8816 of Lecture Notes in Computer Science, pages 63-78. Springer International Publishing.
2. *Riccardo Zese, Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, and Fabiano Aguiari*
   Semantics and inference for probabilistic description logics
   Uncertainty Reasoning for the Semantic Web III, volume 8816 of Lecture Notes in Computer Science, pages 79-99. Springer International Publishing.

- **Conference and Workshop papers**

1. *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*
   An abductive framework for Datalog +/- ontologies
   Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), 2015.
2. *Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, and Giuseppe Cota*

Abductive logic programming for Datalog+/- ontologies
Proceedings of the 30th Italian Conference on Computational Logic (CILC2015), 2015.

3. *Giuseppe Cota, Riccardo Zese, Elena Bellodi, Fabrizio Riguzzi, and Evelina Lamma*
Distributed parameter learning for probabilistic ontologies
25th International Conference on Inductive Logic Programming (ILP 2015), 2015.

4. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota*
Probabilistic inductive constraint logic
25th International Conference on Inductive Logic Programming (ILP 2015), 2015.

5. *Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma*
Structure learning of probabilistic logic programs by mapreduce
25th International Conference on Inductive Logic Programming (ILP 2015), 2015.

6. *Riccardo Zese*
Inference and Learning for Probabilistic Description Logics
Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015), pages 4411-4412, 2015.

7. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
Reasoning with probabilistic ontologies
Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015), pages 4310-4316, 2015.

8. *Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
Logic programming techniques for reasoning with probabilistic ontologies
International Workshop on Ontologies and Logic Programming for Query Answering, 2015. PDF: `http://ontolp.lsis.org/files/pdf/proc-ontolp.pdf#page=13`

9. *Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
   Logic programming techniques for reasoning with probabilistic ontologies
   Proceedings of the Joint Ontology Workshops 2015 Episode 1: The Argentine Winter of Ontology co-located with the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), 2015.
   PDF: `http://ceur-ws.org/Vol-1517/JOWO-15_ontolp_paper_3.pdf`

10. *Giuseppe Cota, Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
    Structure learning with distributed parameter learning for probabilistic ontologies
    Doctoral Consortium of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, pages 75–84, 2015.

11. *Riccardo Zese*
    Learning Probabilistic Description Logics Theories
    Proceedings of the Second Doctoral Workshop in Artificial Intelligence (DWAI 2014), 13th Symposium of the Italian Association for Artificial Intelligence "Artificial Intelligence for Society and Economy", number 1334 in CEUR Workshop Proceedings, pages 13-22, Aachen, Germany, 2014.

12. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
    Learning the parameters of probabilistic description logics
    Late Breaking papers of the 23rd International Conference on Inductive Logic Programming, volume 1187 of CEUR Workshop Proceedings, pages 46-51, 2014.

13. *Riccardo Zese*
    Reasoning with Probabilistic Logics
    ArXiv e-prints 1405.0915v2. An extended abstract / full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), 2014.

14. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
Computing instantiated explanations in OWL DL
Proceedings of the 13th Conference of the Italian Association for Artificial Intelligence, 4-6 December 2013, volume 8249 of Lecture Notes in Artificial Intelligence, pages 397-408, 2013.

15. *Riccardo Zese, Elena Bellodi, Evelina Lamma, and Fabrizio Riguzzi*
A description logics tableau reasoner in Prolog
Proceedings of the 28th Italian Conference on Computational Logic, number 1068 in CEUR Workshop Proceedings, pages 33-47, 2013.

16. *Fabrizio Riguzzi, Evelina Lamma, Elena Bellodi, and Riccardo Zese*
BUNDLE: A reasoner for probabilistic ontologies
7th International Conference on Web Reasoning and Rule Systems, volume 7994 of Lecture Notes in Computer Science, pages 183-197, 2013.

17. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
Parameter learning for probabilistic ontologies
7th International Conference on Web Reasoning and Rule Systems (RR 2013), Mannheim, Germany, July 27-29 2013, volume 7994 of Lecture Notes in Computer Science, pages 265-270, 2013.

18. *Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, and Riccardo Zese*
Epistemic and statistical probabilistic ontologies
Proceedings of the 8th International Workshop on Uncertain Reasoning for the Semantic Web, number 900 in CEUR Workshop Proceedings, pages 3-14, 2012.

19. *Fabrizio Riguzzi, Evelina Lamma, Elena Bellodi, and Riccardo Zese*
Semantics and inference for probabilistic ontologies
Popularize Artificial Intelligence. Proceedings of the AI*IA Workshop and Prize for Celebrating 100th Anniversary of Alan Turing's Birth, volume 860 of CEUR Workshop Proceedings, pages 41-46,2012.

Part II

Description Logics

Chapter 6

Foundations of Description Logics

Knowledge Representation (KR) aims at representing domains (even very complex) in a form that is easily manageable by intelligent systems, i.e. reasoning or learning algorithms, systems able to extract implicit information starting from the explicit one.

In the 70s, work in KR could be divided in two different branches: logic-based formalisms and non-logic-based formalisms. The former exploited logic principles and the predicate calculus for modeling the domain, while the latter described the data by means of ad-hoc representation systems, which were often graphical.

The most common approaches exploited semantic networks or frames, where the knowledge of the domain was modeled via network-shaped structures, in which sets of individuals and their relations were represented by nodes and edges of a graph.

In the following years, network-based systems were prominent, due to their better human-appealing and easier visualization than logic-based systems. Typically, in network-based representations, nodes correspond to concepts (sometimes called frames), that are sets of individuals, while edges depict connections and relationship between concepts. Some systems allow the assignment of attributes to concepts (nodes) or the use of specific nodes to represent complex relationships or particular individuals.

However, they often did not follow precise semantics. This led to systems which behave completely different form each other even though they represent the same information.

The attention gradually shifted to logic systems, in particular based on First Order Logics (FOL for short), for their more precise semantics characterization. By taking advantage of predicate logics, one can effectively represent graphical systems, for example using unary predicate for modeling sets of individuals and binary predicates for representing relationships between individuals. The systems based on networks can be seen as fragment of first order logic and, thus, reasoning processes can be executed exploiting specialized techniques, without necessarily using FOL theorem provers.

The name *terminological systems* appeared for indicating the use of modeling languages for specifying the terminology of the domain of interest. Hereafter, the focus shifted to constructors, which enable to model complex descriptions of concepts. These languages took the name of *concept languages*. Finally, the attention moved more towards the properties of the underlying logic. In these years, the name *Description Logics* (DLs for short) appeared, in order to emphasize the increased attention to logic [3].

In a graphical system, typically the relationships between concepts are *is-a* relations and define a hierarchy of concepts. "A *is-a* B" means that every individual a of A belongs also to B. DL can go beyond these relations and represent more complex relationships between concepts. With respect to properties, also named roles, DLs show more expressive power, for example by allowing the representation of cardinality restrictions or the definition of the range and the domain of the roles.

DLs are a fragment of FOL, thus they can exploit inference techniques that are more performing than those developed for reasoning on FOL. Decidability is a very desirable properties for DLs, which are usually decidable fragments of FOL. DLs are very useful in many domains, such as software engineering, medical diagnosis, digital libraries, databases and Web based informative systems, and generally in all the fields where it is necessary to represent information and to perform inference on it, since they possess nice computational properties such as decidability and/or low complexity.

In the following, Chapter 7 introduces the main characteristics of DLs while Chapter 8 describes the DLs $\mathcal{ALC}$, $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SROIQ}(\mathbf{D})$ showing the differences between them. $\mathcal{ALC}$ is referred as the basis DL while $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SROIQ}(\mathbf{D})$ are the DLs exploited in OWL language. Finally, Chapter 9

briefly describes OWL language and Chapter 10 discusses inference techniques for DLs.

Chapter 7

Description Logics' Characteristics

Description Logics can be defined and classified with respect to three different characteristics:

1. *Basic syntactic units*:
   - *primitive concepts* (also called classes or terms) are abstract groups, sets or collections which gather objects that have common features. They are the fundamental elements and can constitute a hierarchy, where there are super-classes and sub-classes, e.g., Cat and Pet where Cat is the sub-class and Pet is the super-class.
   - *atomic roles* (also named relations) are used for expressing relationships between objects belonging to two possibly different concepts, which represent the domain, the first, and the range, the second.
   - *individuals* (also known as instances) model a specific object of the domain.
2. *Expressiveness*: or expressive power, is determined by the set of constructors allowed for specifying complex concepts and roles.
3. The availability of *inference procedures*: implicit knowledge about concepts, individuals and roles can be automatically inferred by exploiting inference procedures. Especially, subsumption relations between concepts and roles and the "*instance-of*" relations between individuals and concepts are extremely important. An "*instance-of*" relation defines the belonging of an individual to a concept.

DLs detail which kind of *axioms* can be specified. An axiom is expressed for imposing constraints on concepts, instances and roles. The axioms are used during the inference processes for verifying the consistency of the knowledge and for performing inference on it.

7.1 Concept and Role Constructors

The expressiveness of the language determines which constructors are permitted.

Regarding concepts, the most common constructors allowed are conjunction ($\sqcup$), intersection ($\sqcap$) and negation ($\neg$). For negation a differentiation is made between simple concept negation and complex concept negation. A complex concept is a concept defined by a set of concepts (simple or not) combined by constructors. Many DLs define two specific concepts, the *universal top concept* ($\top$), which represents the set of all the individuals and is equivalent to $A \sqcup \neg A$, and the *inconsistent bottom concept* ($\bot$), which represents the empty concept, to which no individuals belong, which is equivalent to $A \sqcap \neg A$. Then, quantification constructors can be added to those already presented. These operators can be classified in unqualified and qualified and are called *existential role restrictions* ($\exists R$ and $\exists R.C$ unqualified and qualified respectively) and *universal role restrictions* ($\forall R$ e $\forall R.C$). In particular, the unqualified constructors correspond to the qualified ones in which the range concept is the top concept, e.g. $\exists R.\top$ e $\forall R.\top$. In some languages, there is also the possibility to define concepts by enumeration, i.e., by listing the admitted individuals, and to define *cardinality restrictions* (again, unqualified and/or qualified). Cardinality restrictions bound the number of individuals related with those contained in the concept under definition. There can be *minimum cardinality restrictions* ($\geq nR$ and $\geq nR.C$), *maximum cardinality restrictions* ($\leq nR$ and $\leq nR.C$) and *exact cardinality restrictions*, often defined in term of the previous two. As for universal and existential qualified role restrictions, in qualified cardinality restrictions the range of the role is constrained. These constructors model the fact that there is a bound on the cardinality of the set of individuals belonging to the specified concept linked to individuals belonging to the complex concept defined by the qualified cardinality restriction. For

example, a truck can be defined as a means of transport with at least four wheels:

$$MeansOfTransport \sqcap \geq 4\ equippedWith.Wheels$$

Some DLs, those that have (**D**) in their name, permit the use of datatype roles, i.e., roles that map an individual to an element of a datatype such as integers, floats, etc. These DLs assume a set of *data values*, and a set of *elementary datatypes*. A *datatype* is an elementary datatype or a finite set of data values. They assume also a set of *datatype predicates*, which present a predefined arity $n \geq 1$. For example, over the integers, ≤ 20 is a unary predicate denoting the set of integers less or equal to 20, and thus $Elephant \sqcap \exists age.\leq 20$ describes elephants younger than 20 years old.

Assuming all the constructors mentioned, we can inductively define complex concepts. Let $\mathbf{A}$, $\mathbf{R_A}$, $\mathbf{R_D}$ and $\mathbf{I}$ be disjoint sets of *atomic concepts*, *abstract roles*, *datatype roles* and *individuals*, respectively, then the following are concepts:

- $\top$
- $\bot$
- every $A \in \mathbf{A}$
- for every finite set $\{a_1, ..., a_n\} \in \mathbf{I}$ of individuals names, $\{a_1, ..., a_n\}$ is a concept called *nominal*

If C and B are concepts and $R \in \mathbf{R_A}$ then the following are concepts as well:

- $(C \sqcap B)$, the intersection of two concepts
- $(C \sqcup B)$, the union of two concepts
- $\neg C$, the complement 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 of a concept by a role

- $\geq nR$ and $\leq nR$ for an integer $n \geq 0$, unqualified number restriction on a role
- $\geq nR.C$ and $\leq nR.C$ for an integer $n \geq 0$, qualified number restriction on a role

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

- $\exists T.D$, the datatype existential restriction of a concept by a role
- $\forall T.D$, the datatype universal restriction of a concept by a role

Some logics with high expressive power allow also role constructors, such as union, composition ($\circ$) for defining chains of roles, transitive closure, functional roles and inverse roles.

7.2 Family of DLs

The expressiveness is indicated using a well-established naming convention which specifies what is allowed and what is not allowed in the DL. The naming scheme can be summarized as follows:

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

where

- $\mathcal{ALC}$ is the abbreviation of *attributive language with complements*. It is often considered as the base language and allows atomic negation ($\neg$), conjunction ($\sqcup$) and intersection ($\sqcap$) as well as universal ($\forall R.C$) and (limited) existential ($\exists R.C$) quantifiers. It allows also the negation of complex concepts ($\mathcal{C}$).
- $\mathcal{FL}$ is the contraction of *frame based description language*. It allows concept intersections, universal restrictions, limited existential quantifications and role restrictions. $\mathcal{FL}$ has two sublanguages: $\mathcal{FL}^-$, obtained by disallowing the use of role restrictions, and $\mathcal{FL}_o$, that is a sublanguage of $\mathcal{FL}^-$ obtained by disallowing limited existential quantifications.

- $\mathcal{EL}$ 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 extention which permits the use of role inclusion axioms together with the concept inclusion axioms already allowed by $\mathcal{EL}$. $\mathcal{EL}^{++}$ is an alias for $\mathcal{ELRO}$.
- $\mathcal{S}$ denotes the logic $\mathcal{ALC}$ extended with the possibility to define 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.
- $\mathcal{O}$ 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 *Beatles* can be $\{john, paul, ringo, george\}$.
- $\mathcal{I}$ enables the definition of inverse roles.
- $\mathcal{F}$ indicates the possibility of defining functional role statements.
- $\mathcal{E}$ means that the DL features full existential quantifications.
- $\mathcal{U}$ allows union between concepts.
- $\mathcal{N}$ means that the definition of unqualified cardinality restrictions is allowed, i.e., $\leq R$ and $\geq R$.
- $\mathcal{Q}$ means that qualified cardinality restrictions can be defined, i.e., $\leq R.C$ and $\geq R.C$.
- (**D**) allows datatype properties.

7.3 Knowledge Base

In systems where the knowledge management is one of the main objectives, it is necessary to bear in mind two important aspects:

- the characterization and the definition of the knowledge base, which has to be formally and precisely specified;
- the definition of environments and frameworks which allow processing, querying and management of the knowledge base.

These two features are interconnected, as the choice of how to define knowledge base must take into account which reasoning services ought to be provided, insofar as they depend on how the knowledge is specified.

A knowledge base (KB for short) contains two kinds of information, *intensional knowledge* and *extensional knowledge*. The first, generally, is divided into *Terminological Box* (TBox) and *Role Box* (RBox) and models general information about the domain, normally contains immutable information and is built through statements which describe the main properties of roles and concepts. The latter, called *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.

7.3.1 TBox

The TBox contains axioms regarding concepts. They can be grouped into *definitions* and *subsumptions*.

Definitions 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

$$Parent \equiv Mother \sqcup Father$$

Subsumptions introduce a hierarchy among concepts. These axioms are also called *concept inclusion axioms* and specify a *is-a* relationship between two different concept. For example, we can state that a man is a person as

$$Man \sqsubseteq Person$$

Definition can be expressed as subsumptions as $C \equiv D$ is equivalent to $C \sqsubseteq D$ and $D \sqsubseteq C$. A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts.

7.3.2 RBox

The RBox is a set of axioms that describes the roles contained in the KB. A *role* is either an atomic role $R \in \mathbf{R_A}$ or the inverse R^- of an atomic role $R \in \mathbf{R_A}$. We use $\mathbf{R_A}^-$ to denote the set of all inverses of roles in $\mathbf{R_A}$.

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*.

Role inclusion axioms are of the form $R \sqsubseteq S$, where $R, S \in \mathbf{R_A} \cup \mathbf{R_A}^-$ or $R, S \in \mathbf{R_D}$. We call *role equivalence axiom* the statement $R \equiv S$, 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:

$$fatherOf \circ fatherOf \sqsubseteq grandParentOf$$

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.

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 that, then the KB is inconsistent

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 RBox, e.g. $\mathcal{ALC}$.

7.3.3 ABox

An *ABox* (Assertional Box) contains information about the specific domain of the problem. It introduces the individuals, that are instances of classes in the domain, and specifies the properties of the individuals in terms of concepts and roles. 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 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.

7.4 Semantics

Generally a DL is assigned a semantics following a *set-theoretic* approach where every concept is interpreted as a set of individuals and every role as a set of

pairs of individuals. A knowledge base $\mathcal{K}$ is usually assigned a semantics in terms of interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where

- $\Delta^{\mathcal{I}}$ is a non-empty *domain*, which contains all the individuals of the domain,
- $\cdot^{\mathcal{I}}$ is the *interpretation function*, that assigns an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to each $a \in \mathbf{I}$, a subset of $\Delta^{\mathcal{I}}$ to each $C \in \mathbf{A}$ and a subset 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, concepts and 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 $\mathcal{I}$. 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 of arity n a relation over $\Delta^{\mathcal{D}}$ of arity n.

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 $\cdot^{\mathcal{I}}$ is extended to complex concepts (where $R^{\mathcal{I}}(x) = \{y|(x,y) \in R^{\mathcal{I}}\}$, $R^{\mathcal{I}}(x, C) = \{y|\langle x, y\rangle \in R^{\mathcal{I}}, y \in C^{\mathcal{I}}\}$), $\#X$ denotes the cardinality of the set X and $T^{\mathcal{I}}(x) = \{y|y \in \Delta^{\mathcal{D}}, (x,y) \in T^{\mathcal{I}}\}$) as follows:

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, the set of all the individuals of the domain
- $\bot^{\mathcal{I}} = \emptyset$, the empty concept
- $(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$, each individual is a member of both C_1 and C_2

- $(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$, each individual is a member of at least one of the concepts C_1 and C_2
- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$, the set of individuals that do not belong to the concept C
- $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \subseteq C^{\mathcal{I}}\}$, each individual that belongs to this concept is related by the role R *only* to individuals that belong to concept C
- $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \cap C^{\mathcal{I}} \neq \emptyset\}$, each individual that belongs to this concept is related through R to *at least one* individual that belongs to C
- $(R^-)^{\mathcal{I}} = \{(y, x) | (x, y) \in R^{\mathcal{I}}\}$
- $(R_1 \circ ... \circ R_n)^{\mathcal{I}} = R_1^{\mathcal{I}} \circ ... \circ R_n^{\mathcal{I}}$
- $\{a\}^{\mathcal{I}} = \{a^{\mathcal{I}}\}$
- $(\geq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x) \geq n\}$
- $(\geq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x, C) \geq n\}$
- $(\leq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x) \leq n\}$
- $(\leq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x, C) \leq n\}$
- $(\forall T.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | T^{\mathcal{I}}(x) \subseteq D^{\mathcal{I}}\}$
- $(\exists T.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | T^{\mathcal{I}}(x) \cap D^{\mathcal{I}} \neq \emptyset\}$

Given a specific interpretation $\mathcal{I}$, we can determine the *satisfaction* of an axiom E, i.e. whether the axiom holds (is true) with respect to $\mathcal{I}$. The satisfaction, denoted by $\mathcal{I} \models E$, is defined as follows:

1. A concept inclusion axiom $\mathcal{I} \models C \sqsubseteq D$ is satisfied in $\mathcal{I}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$,
2. A concept assertion axiom $\mathcal{I} \models a : C$ is satisfied in $\mathcal{I}$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$,
3. A role assertion axiom $\mathcal{I} \models (a, b) : R$ is satisfied by $\mathcal{I}$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$,

4. An equality axiom $\mathcal{I} \models a = b$ holds in $\mathcal{I}$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$,

5. A inequality axiom $\mathcal{I} \models a \neq b$ is satisfied by $\mathcal{I}$ iff $a^{\mathcal{I}} \neq b^{\mathcal{I}}$,

6. A transitivity axiom $\mathcal{I} \models Trans(R)$ is satisfied by $\mathcal{I}$ iff $R^{\mathcal{I}}$ is transitive,

7. A role inclusion axiom $\mathcal{I} \models R \sqsubseteq S$ is satisfied by $\mathcal{I}$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$, this condition can be generalized for role chains as follows

   (a) $\mathcal{I} \models R_1 \circ ... \circ R_n \sqsubseteq S$ is satisfied by $\mathcal{I}$ iff $(R_1 \circ ... \circ R_n)^{\mathcal{I}} \subseteq S^{\mathcal{I}}$,

8. A datatype role assertion axiom $\mathcal{I} \models (a, v) : T$ for a data value v is satisfied by $\mathcal{I}$ iff $(a^{\mathcal{I}}, v^{\mathcal{D}}) \in T^{\mathcal{I}}$.

$\mathcal{I}$ *satisfies* a set of axioms $\mathcal{E}$, denoted by $\mathcal{I} \models \mathcal{E}$, iff $\mathcal{I} \models E$ for all $E \in \mathcal{E}$. 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 $\mathcal{K}$ satisfies also E. For example, a concept C is *satisfiable* relative to $\mathcal{K}$ iff there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$.

A translation of DL into First-Order Logic with Counting Quantifiers is given in the following as an extension of the one given in [132]. The translation uses two functions π_x and π_y that map concept expressions to logical formulas,

Table 7.1: Correspondence between DL axioms and their translation into predicate logic. Functions π_x and π_y are exploited to translate the concepts contained in the axioms.

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

where π_x is given by

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

and π_y is obtained from π_x by replacing x with y and vice-versa. Table 7.1 shows the translation of the most used axiom types of DL knowledge bases.

Each DL is *decidable* if the problem of checking the satisfiability of a KB is decidable. In particular, a DL is decidable iff there are no number restrictions on transitive roles and on roles that has transitive subroles.

Role chains introduce some issues too:

- Arbitrary property chain axioms lead to undecidability. For ensuring decidability the following restriction must be imposed: the set of property chain axioms must be *regular*, i.e., the set has to contain only property chain axioms of the following forms:
  $R \circ R \sqsubseteq R$
  $S^- \sqsubseteq R$
  $S_1 \circ S_2 \circ ... \circ Sn \sqsubseteq R$
  $R \circ S1 \circ S2 \circ ... \circ Sn \sqsubseteq R$
  $S1 \circ S2 \circ ... \circ Sn \circ R \sqsubseteq R$
  in which, given a strict linear order $\prec$, $S_i \prec R$ for all $i = 1, 2, ..., n$.

- The combination of property chain axioms with cardinality constraints may lead to undecidability. For ensuring decidability, only simple properties are allowed in cardinality expressions. A property is simple iff it is not a super property of a property chain, i.e., R is simple iff there is not any property chain axiom of the form $S_1 \circ S_2 \circ ... \circ S_n \sqsubseteq R$ with $n > 1$, nor any subproperty axiom $S \sqsubseteq R$ with S non simple.

Chapter 8

Significant Examples of Description Logics

DLs differ for the set of constructors they admit, both for roles and for concepts. In this chapter we will show, in the light of what has already been said, the difference between three different DLs that are relevant for the work presented in this Thesis: $\mathcal{ALC}$, $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SROIQ}(\mathbf{D})$.

As seen before, $\mathcal{ALC}$ is an alias for Attributive Language with Complement. It features negation, conjunction, intersection and both universal and existential quantification, together the universal top concept $\top$ and the inconsistent bottom concept $\bot$. It allows concept inclusion axioms, concept and role membership axioms, equality and inequality axioms. $\mathcal{ALC}$ DL does not permit the definition of axioms concerning roles, therefore, a *knowledge base* $\mathcal{K}$ consists only of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$, i.e. $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where the ABox does not contain datatype role membership axioms.

The interpretation function $\cdot^{\mathcal{I}}$ for $\mathcal{ALC}$ maps the following complex concepts: (1) $\top$, (2) $\bot$, (3) $C_1 \sqcap C_2$, (4) $C_1 \sqcup C_2$, (5) $\neg C$, (6) $\forall R.C$ and (7) $\exists R.C$. The satisfaction of an axiom E is defined in Section 7.4 in points 1-5, thus without definitions concerning roles since the absence of the RBox.

The DL $\mathcal{SHOIN}(\mathbf{D})$ underlies the first definition of OWL DL. It adds to $\mathcal{ALC}$ role transitivity (denoted by the $\mathcal{S}$), role hierarchy ($\mathcal{H}$), nominals ($\mathcal{O}$), role inverses ($\mathcal{I}$), unqualified number restrictions ($\mathcal{N}$) and datatype roles ($\mathbf{D}$). Together with the constructor that $\mathcal{ALC}$ permits, $\mathcal{SHOIN}(\mathbf{D})$ also adds roles axioms on abstract and datatype roles. A $\mathcal{SHOIN}(\mathbf{D})$ KB $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$

consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$ and an ABox $\mathcal{A}$, where the RBox $\mathcal{R}$ consists of a finite set of transitivity axioms and role inclusion axioms, while the ABox is a finite set of axioms as specified in Subsection 7.3.3.

The mapping $\cdot^{\mathcal{I}}$ for $\mathcal{SHOIN}(\mathbf{D})$ is extended to all new complex concepts accepted, that are: (8) R^{-}, (9) $\{a\}$, (10) $\geq nR$, (11) $\leq nR$ and (12) $\forall T.D$ and $\exists T.D$ for datatype roles. The *satisfaction* of an axiom E takes into account the definitions given for $\mathcal{ALC}$ plus the ones regarding RBox axioms except for the general definition of roles chain inclusion axioms (Section 7.4, point 7.a). $\mathcal{SHOIN}(\mathbf{D})$ is decidable iff there are no number restrictions on transitive roles and on roles that has transitive subroles.

Finally, $\mathcal{SROIQ}(\mathbf{D})$ [62] adds to $\mathcal{SHOIN}(\mathbf{D})$ qualified cardinality restrictions and complex role inclusions, plus some new features such as roles disjunction, reflexive, irreflexive and antisymmetric roles and the definition of a universal role, similar to $\top$ for the concepts. The interpretation function and the definition of satisfiability correspond to those described in Section 7.4.

Example 1. *We would like to model the class of individuals who love Italian cars, named ItalianCarsLover. Using $\mathcal{ALC}$, we give the following definition of the concept: "an Italian cars lover is a person who has bought for himself at least an Italian car and all cars he has bought are Italian cars". In $\mathcal{ALC}$, this sentence can be represented as:*

$$\begin{aligned} ItalianCarsLover \equiv \exists hasBoughtForHimself.ItalianCars \sqcap \\ \forall hasBoughtForHimself.ItalianCars \end{aligned} \tag{8.1}$$

where the class ItalianCar can be modeled as the union of cars branded "Fiat" and cars branded "Alfa Romeo":

$$ItalianCar \equiv FiatBrandedCars \sqcup AlfaRomeoBrandedCars$$

We know that Mario loves Italian cars, in fact all cars he has possessed are Italian, but his first car was a present from his parents. Thus

$$\begin{aligned} mario : \exists hasPossessed.ItalianCars \sqcap \forall hasBoughtForHimself.ItalianCars \\ \sqcap \forall hasPossessed.ItalianCars \end{aligned}$$

Following the definition of (8.1), mario is not an Italian branded cars lover. For solving this problem we could note that if one buys something, he/she owns it. With $\mathcal{ALC}$ we cannot specify this kind of information, we have to use more expressive languages, such as $\mathcal{SHOIN}(\mathbf{D})$, where we can specify that

$$hasBoughtForHimself \sqsubseteq hasPossessed$$

and so modify (8.1) to:

$$\begin{aligned} ItalianCarsLover \equiv\ & \exists hasPossessed.ItalianCars \sqcap \\ & \forall hasPossessed.ItalianCars \end{aligned} \tag{8.2}$$

In this way Mario : ItalianCarsLover now is true. We also know that Mauro loves Italian cars too, in fact he has owned more than 5 Italian cars, but one of his first cars was not an Italian car. In order to include Mauro in the class of ItalianCarsLover we could change the definition of Italian branded cars lover as "the set of individuals who have owned at least 5 Italian cars", therefore (8.2) becomes:

$$ItalianCarsLover \equiv \geq 5\ hasPossessed.ItalianCars \tag{8.3}$$

Note that in (8.3) we need qualified *number restrictions, allowed by $\mathcal{SROIQ}(\mathbf{D})$ but not by $\mathcal{SHOIN}(\mathbf{D})$.*

Chapter 9

OWL: the Web Ontology Language

In 2004, the W3C standardized a language that is more expressive and richer than RDF and RDFS, called Web Ontology Language, abbreviated in OWL. There are many stories behind the choice of this acronym, which does not correspond to the extended name. It was hypothesized that the acronym was a reference to an Artificial Intelligence project called One World Language, an early attempt studied in the 70s for defining a universal language for encoding information for computers. Another legend has it that the choice was due to the character Owl from Winnie the Pooh, who misspells his name as "WOL", the real acronym for Web Ontology Language. But the most accredited was due to Guus Schreiber, one of the co-chairs of the Web Ontology Work Group of the W3C created to standardize OWL. Guus Schreiber, regarding the acronym, asserted "Why not be inconsistent in at least one aspect of a language which is all about consistency?".

The first version of OWL defines three different sublanguages of increasing complexity and with different level of expressiveness:

OWL Lite based on the Description Logic $\mathcal{SHIF}(\mathbf{D})$, supports classification hierarchies and simple constraints. For example, it admits cardinality restrictions with cardinality values of 0 or 1 only. OWL Lite was intended to be easily computable since it is targeted to thesauri and taxonomies.

OWL DL originally based on $\mathcal{SHOIN}(\mathbf{D})$, it allows all the constructors and

the axioms permitted by that DL. OWL DL is a language meant to be decidable and computationally complete while maintaining the maximum expressiveness possible.

OWL Full a highly expressive semantics. For example, classes can be seen as both collections of individuals and single individuals. OWL Full is not decidable and the support to complete reasoning is unlikely.

In 2008 the W3C OWL Working Group published the specifications of the follower of OWL, called OWL2, based on $\mathcal{SROIQ}(\mathbf{D})$ [60]. OWL2 is also equipped with three different profiles:

- **OWL 2-EL**, it was expressly defined for application with very large numbers of classes and/or properties. It corresponds with $\mathcal{EL}^{++}$ for which basic reasoning problems can be preformed in polynomial time in the size of the ontology. The definition of this profile was motivated by the need to model large medical and biochemical ontologies, such as Gene Ontology[1] or SNOMED-CT[2] where there are thousands of classes defined.

- **OWL 2-QL**, a fragment offering a simplified support to queries in order to manage large number of instance data. It allows to maintain data in relational databases where reasoning can be performed by means of query languages. This profile allows defining herarchies between classes and properties together with inverse properties but disallows, for example, the use of universal quantifiers.

- **OWL 2-RL**, an OWL subset meant to handle rules, such as *if-then-else* constructs. This profiles makes use of standard rule languages and queries can be solved using rule-based reasoning engines. It was defined for introducing more expressivity in RDF(S) applications without the need to re-define data exploiting other DL languages.

[1] `http://geneontology.org/`
[2] `http://www.ihtsdo.org/snomed-ct`

Chapter 10

Inference in Description Logics

In the previous chapter we presented languages for modeling information and defining Knowledge Bases (KBs). After the creation of a KB, it is important to provide systems able to execute inference over the modeled information and systems able to automatically learn or repair the specified knowledge. In this chapter the inference problem is addressed, which aims at extracting implicit information from those explicit in the KBs. Usually, inference systems are able to answer queries given by users by testing whether the queries hold w.r.t. the input information and in many cases also to provide explanations for the result.

Here we concentrate on the problem of finding explanations for a query that has been investigated by various authors [56, 68, 70, 71, 133]. Also called *axiom pinpointing* by Schlobach and Cornet [133], it is considered as a non-standard reasoning service useful for tracing derivations and debugging ontologies. In particular, the authors of [133] define *minimal axiom sets* or *MinAs* for short.

Definition 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 [133]. ALL-MINAS$(Q, \mathcal{K})$ is the set of all MinAs for query Q in the knowledge base $\mathcal{K}$.

Problem: MIN-A-ENUM

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 $\mathcal{K}$.

Instead of finding ALL-MINAS$(Q, \mathcal{K})$ for queries, in [4, 5] Baader and Peñaloza presented the problem of finding a *pinpointing formula* which compactly encodes the set of all MinAs. The pinpointing formula is a monotone Boolean formula built using Boolean variables corresponding to an axiom of the KB and the conjunction and disjunction connectives. Let assume that each axiom E of a KB $\mathcal{K}$ is associated with a propositional variable, indicated with $var(E)$. The set containing all propositional variables is indicated with $var(\mathcal{K})$ while the set of propositional variables that are true is indicated with ν, which is called *valuation* of a monotone Boolean formula. For a valuation $\nu \subseteq var(\mathcal{K})$, let $\mathcal{K}_\nu := \{t \in \mathcal{K} | var(t) \in \nu\}$.

Definition 2 (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 $\nu \subseteq var(\mathcal{K})$ it holds that $\mathcal{K}_\nu \models Q$ iff ν satisfies ϕ.*

In Lemma 2.4 of [5], the authors proved that we can obtain all MinAs from a pinpointing formula by transforming the formula into Disjunctive Normal Form (DNF) and removing disjuncts implying other disjuncts.
The example below illustrates the difference between ALL-MINAS$(Q, \mathcal{K})$ and the pinpointing formula.

Example 2 (Pinpointing formula)**.** *The following KB is inspired by the ontology* `people+pets` *[101]:*

$$\begin{aligned}
&\exists hasAnimal.Pet \sqsubseteq NatureLover\\
&\mathit{fluffy} : Cat\\
&tom : Cat\\
&Cat \sqsubseteq Pet\\
&(kevin, \mathit{fluffy}) : hasAnimal\\
&(kevin, tom) : hasAnimal
\end{aligned}$$

It states that every individual that owns an animal that is a pet is a nature lover and that fluffy and tom are cats. Moreover, all cats are pets and kevin owns the animals fluffy and tom. We associate Boolean variables with axioms as follows: $F_1 = \exists hasAnimal.Pet \sqsubseteq NatureLover$, $F_2 = (kevin, \mathit{fluffy}) : hasAnimal$, $F_3 = (kevin, tom) : hasAnimal$, $F_4 = \mathit{fluffy} : Cat$, $F_5 = tom : Cat$ and $F_6 = Cat \sqsubseteq Pet$. Let $Q = kevin : NatureLover$ be the query, then

ALL-MINAS$(Q, \mathcal{K}) = \{\{F_2, F_4, F_6, F_1\}, \{F_3, F_5, F_6, F_1\}\}$, *while the pinpointing formula is* $((F_2 \wedge F_4) \vee (F_3 \wedge F_5)) \wedge F_6 \wedge F_1$. *It is easy to see that the pinpointing formula is equivalent to* $((F_2 \wedge F_4 \wedge F_6 \wedge F_1) \vee (F_3 \wedge F_5 \wedge F_6 \wedge F_1))$ *that corresponds to* ALL-MINAS$(Q, \mathcal{K})$.

In the following we define how the two different approaches can be implemented by considering a state-of-art system, called Pellet [136], which is at the basis of BUNDLE, one of the system presented in this Thesis presented in Chapter 15.

10.1 Approaches to Compute Explanations

The most common approach to solve MIN-A-ENUM problem uses a *tableau algorithm* [134] to decide whether an axiom is entailed or not by a KB. Such an algorithm works by refutation: it tries to prove whether a concept C is unsatisfiable by showing that the assumption of non-empty C leads to contradiction. This is done by assuming that C has an instance and by trying to build a model for the KB. If no model can be built, then C is unsatisfiable, otherwise the model is a counter example for the unsatisfiability of C.

The algorithm works on *completion graphs* also called *tableaux*. A tableau represents an ABox in which each node a represents an individual a, labeled with the set of concepts $\mathcal{L}(a)$ it belongs to. Each edge $\langle a, b\rangle$ in the graph is labeled with the set of roles $\mathcal{L}(\langle a, b\rangle)$ to which the couple (a, b) belongs. The reasoner starts from a tableau that contains the information of the ABox of the KB and the negation of the query axiom. To prove the unsatisfiability of a concept C, an anonymous individual a is assumed to be in C, thus C is assigned to the label of a. The entailment of any type of axiom by a KB can be checked by means of the tableau algorithm. For example, the axiom $C \sqsubseteq D$ can be proved by showing that $C \sqcap \neg D$ is unsatisfiable. Similarly, checking whether $a : C$ holds can be done by asserting that a is an instance of $\neg C$ and then showing that this leads to contradiction.

After the initialization of the tableau, the algorithm repeatedly applies a set of consistency preserving *tableau expansion rules* until a contradiction, usually called *clash*, is detected or a clash-free graph is found to which no more rules are applicable. Some of the rules are non-deterministic, i.e., they generate

a finite set of tableaux. Thus the algorithm keeps a set of tableaux that is consistent if there is any tableau in it that is consistent, i.e., that is clash-free.

Following [70], MIN-A-ENUM can be solved either with reasoner dependent (glass-box) approaches or reasoner independent (black-box) approaches. Glass-box approaches are built on existing tableau-based decision procedures and modify the internals of the reasoner. Black-box approaches use the DL reasoner solely as a subroutine and the internals of the reasoner do not need to be modified.

The techniques of [56, 69, 70, 71] for axiom pinpointing have been integrated into the Pellet reasoner [136]. By default, Pellet solves MIN-A-ENUM with a hybrid glass/black-box approach: it finds all the MinAs by starting from a single MinA found using a modified tableau algorithm (glass-box) and then running a black box method, the *hitting set tree* algorithm, exploiting in turn the modified tableau. Roughly speaking, the hitting set algorithm recursively selects an axiom of the MinA, removes it from the KB and looks for alternative explanations. It repeats this process until the query is not entailed, building the set of all explanations.

Differently, in [5] Baader and Peñaloza illustrated how a standard tableau algorithm can be modified to build a pinpointing formula instead of MinAs.

Both approaches were implemented in the three different systems presented in Part IV of this Thesis. BUNDLE is based on Pellet and uses it for solving the MIN-A-ENUM problem, TRILL implements BUNDLE's tableau algorithm by exploiting a different programming paradigm, and finally TRILL^P implements the approach of [5]. In the following, we illustrate the two different approaches to solve MIN-A-ENUM.

10.1.1 Solving MIN-A-ENUM: The Standard Definition

In this Section we present the solution for MIN-A-ENUM that exploits a *tableau* algorithm.

In the following we describe the tableau algorithm used by Pellet for solving MIN-A-ENUM. The algorithm is shown in Algorithm 1. Pellet finds a MinA by modifying the tableau expansion rules so that a *tracing function* τ is updated as well [56, 68, 69]. τ associates sets of axioms with events in the derivation.

Formally, a *completion graph* for a knowledge base $\mathcal{K}$ is a tuple $G =$

Algorithm 1 Tableau algorithm executed by Pellet.

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

$(V, E, \mathcal{L}, \dot{\neq})$ in which (V, E) is a directed graph. Each node $a \in V$ is labeled with a set of concepts $\mathcal{L}(a)$ and each edge $e = \langle a, b \rangle$ is labeled with a set $\mathcal{L}(e)$ of role names. The binary predicate $\dot{\neq}$ is used to specify the inequalities between nodes.

In order to manage non-determinism, 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 asserted in the knowledge base, labeled with the nominal $\{a\}$ plus all concepts C such that $a : C \in \mathcal{K}$, and an edge $e = \langle a, b \rangle$ labeled with R for each assertion $(a, b) : R \in \mathcal{K}$.

At each step of the algorithm, an expansion rule is applied to a completion graph G from T: G is removed from T, the rule is applied and the results are inserted in T. The rules used by Pellet are shown in Figure 10.1. For example, if the rule $\rightarrow \sqcap$ is applied, a concept $C \sqcap D$ in the label of a node a causes C and D to be added to $\mathcal{L}(a)$, because the individual that a represents must be an instance of both C and D.

If a *non-deterministic* rule is applied to a graph G in T, then G is replaced by the resulting set of graphs. For example, if the disjunction $C \sqcup D$ is present in the label of a node, the rule $\rightarrow \sqcup$ generates two graphs, one in which C is added to the node's label and the other in which D is added to the node's label.

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

1. $Add(C, a)$, the addition of a concept C to $\mathcal{L}(a)$;
2. $Add(R, \langle a, b\rangle)$, the addition of a role R to $\mathcal{L}(\langle a, b\rangle)$;
3. $Merge(a, b)$, the merging of the nodes a, b;
4. $\dot{\neq}(a, b)$, the addition of the inequality $a \dot{\neq} b$ to the relation $\dot{\neq}$;
5. $Report(g)$, 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 C and $\neg C$ are present in the label of a node, i.e. $\{C, \neg C\} \subseteq \mathcal{L}(a)$;
- a couple $(Merge(a, b), \dot{\neq}(a, b))$, where the events $Merge(a, b)$ and $\dot{\neq}(a, b)$ belong to $\mathcal{E}$.

When a clash is detected in a completion graph G, the algorithm stops applying rules to G. The algorithm terminates when every completion graph in T contains a clash or no more expansion rules can be applied to it. At this point, the algorithm checks all the completion graphs in the final set T, if all of them contain a clash, the algorithm cannot find a model for the concept and returns *unsatisfiable*. Otherwise, it collect all the models from all the clash-free completion graph in T and returns *satisfiable*. Both *soundness* and *completeness* of the algorithm rely in these observations.

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

Deterministic rules:

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

$\rightarrow$ *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**
$Add((\neg C \sqcup D), a)$, $\tau((\neg C \sqcup D), a) := \{C \sqsubseteq D\}$

$\rightarrow \sqcap$ $(*)$: **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**
$Add(\{C_1, C_2\}, a)$, $\tau(C_i, a) := \tau((C_1 \sqcap C_2), a)$

$\rightarrow \exists$ $(*)$: **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, $Add(S, \langle a, b\rangle)$, $Add(C, b)$
$\tau(C, b) := \tau((\exists S.C), a)$, $\tau(S, \langle a, b\rangle) := \tau((\exists S.C), a)$

$\rightarrow \forall$ $(*)$: **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**
$Add(C, b)$, $\tau(C, b) := \tau((\forall S.C), a) \cup \tau(S, \langle a, b\rangle)$

$\rightarrow \forall^+$: **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**
$Add(\forall R.C, b)$, $\tau((\forall R.C), b) := \tau((\forall S.C), a) \cup \tau(R, \langle a, b\rangle) \cup \{Trans(R)\} \cup \{R \sqsubseteq S\}$

$\rightarrow \geq$: **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 $b_1, ..., b_n$, $Add(S, \langle a, b_i\rangle); \neq(b_i, b_j)$
$\tau(S, \langle a, b_i\rangle) := \tau((\geq nS), a)$, $\tau(\neq(b_i, b_j)) := \tau((\geq nS), a)$

$\rightarrow O$: **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**
$\tau(Add(C_i, \mathcal{L}(b))) := \tau(Add(C_i, \mathcal{L}(a))) \cup \tau(Merge(a, b))$
(similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(b)$)

Non-deterministic rules:

$\rightarrow \sqcup$ $(*)$: **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 $i \in \{1, 2\}$

$\rightarrow \leq$: **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)

Figure 10.1: Pellet tableau expansion rules; the subset of rules marked by $(*)$ is employed by TRILL^P presented in Chapter 17.

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 $\tau(R, \langle a, b\rangle)$ 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 10.1) add axioms to values of τ.

For ensuring the termination of the algorithm, a special condition known as *blocking* [68] 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. 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-neighbor* of a node x if y is a successor of x and $R \in \mathcal{L}(\langle x, y\rangle)$, where $R \in \mathbf{R}$. 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:

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.

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

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 $S = \bigcup_{i\in\{1,...,n\}} s_{g_i} \setminus \{\neg C(a)\}$ where a is the anonymous individual initially

Algorithm 2 Black-Box pruning algorithm.

```
1: function BlackBoxPruning(C, S)
2:     Input: Q (the concept to be tested for satisfiability)
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 C is satisfiable w.r.t. S then
8:             S ← S ∪ {E}
9:         end if
10:    end for
11:    return S
12: end function
```

Algorithm 3 SingleMinA algorithm.

```
1: function SingleMinA(C, K)
2:     Input: Q (the concept to be tested for satisfiability)
3:     Input: K (the knowledge base)
4:     Output: S (a MinA for the unsatisfiability of Q w.r.t. K) or null
5:     S ←Tableau(C, K)
6:     if S = null then
7:         return null
8:     else
9:         return BlackBoxPruning(C, S)
10:    end if
11: end function
```

assigned to $\neg C$. 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.

Thus S is pruned using a black box approach [68] called BlackBoxPruning and shown in Algorithm 2. This algorithm executes a loop on S: in each iteration it removes an axiom from S and checks whether the concept C turns satisfiable w.r.t. the reduced verison of S. If the concept turns satisfiable, the axiom is reinserted into S as the axiom extracted is responsible for the unsatisfiability of C. Vice-versa, if the concept still remains unsatisfiable, the removed axiom is irrelevant and can be removed from 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 3, first executes Tableau and then BlackBoxPruning.

The output S of SingleMinA is guaranteed to be a MinA, as established by the following theorem, where All-MinAs$(Q, \mathcal{K})$ stands for the set of MinAs

for Q in which the corresponding concept C is unsatisfiable:

Theorem 1. *[68] Let Q be a query and C be the unsatisfiable concept w.r.t. $\mathcal{K}$ corresponding to Q and let S be the output of the algorithm* SingleMinA *with input C, $\mathcal{K}$, then $S \in$ All-MinAs$(Q, \mathcal{K})$.*

Proof. We need to prove that the output of Tableau S (before it is pruned) includes at least one explanation, i.e., C is unsatisfiable w.r.t. S. Let $\mathcal{E}$ be the sequence of events generated by Tableau with inputs C and $\mathcal{K}$. Now suppose Tableau is used with input C and S and let T', $\mathcal{E}'$ be the corresponding sets of completion graphs and events generated. For each event $\varepsilon_i \in \mathcal{E}$, it is possible to perform ε_i in the same sequence as before. This is because, for each event ε_i, the set of axioms in $\mathcal{K}$ responsible for ε_i have been included in the output S by construction of the tracing function τ in Figure 10.1. Thus, given $\mathcal{E}' = \mathcal{E}$, a clash occurs in each of the completion graphs in T' and the algorithm finds C unsatisfiable w.r.t. S. □

SingleMinA returns a single MinA. To compute all MinAs, Pellet uses Reiter's *hitting set algorithm* [112]. In [112], Reiter developed a general theory of diagnosis where a system to be diagnosed is a pair $(SD, COMPONENTS)$ where SD is a set of first-order sentences which describe the system and $COMPONENTS$ is a finite set of constants. A set of observation OBS is then associated to 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 $\Delta \subseteq COMPONENTS$ such that

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

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 $\{c_1, ...c_n\} \subseteq COMPONENTS$ 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. $COMPONENTS - \Delta$ 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 $C_i \cap HS \neq \emptyset$ 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 unsatisfiability, can be found by exploiting an algorithm that generates minimal hitting sets [68, 70]. The universal set corresponds to the total set of axioms in the KB, and an explanation (for a particular concept unsatisfiability) corresponds to a single conflict set.

Reiter's algorithm constructs a labeled tree called *Hitting Set Tree* (HST). In a HST a node v is labeled with OK, with X or with a set $\mathcal{L}(v) \in CS$ and an edge e is labeled with an element of U. Let $H(v)$ be the set of edge labels on the path from the root of the HST to node v. For each element $E \in \mathcal{L}(v)$, v has a successor w connected to v by an edge with E in its label. If $\mathcal{L}(v) = OK$, then $H(v)$ is a hitting set for CS.

The algorithm, described in detail in [68] and shown in Algorithm 4, starts from a MinA S and initializes an HST $\mathbf{T} = (V, E, \mathcal{L})$ with S as the label of its root node, i.e. $V = \{v\}, E = \emptyset, \mathcal{L}(v_0) = S$. Then it selects an arbitrary axiom E in S, it removes it from $\mathcal{K}$, generating a new knowledge base $\mathcal{K}' = \mathcal{K} - \{E\}$, and it tests the unsatisfiability of C w.r.t. $\mathcal{K}'$ by invoking SingleMinA. If C is unsatisfiable, we obtain a new explanation for the unsatisfiability of C that is a new explanation for the corresponding query Q. A new node w labeled with this new explanation and a new edge $\langle v, w \rangle$ labeled with the axiom E are added in the tree. When the unsatisfiability test returns negative (the concept turns satisfiable), the algorithm terminates to repeat this process and labels the new node w with OK, making it a leaf. $H(w)$ represents a hitting set and the path from the root to w is a *hitting set path*. The algorithm also uses previous results to eliminate useless unsatisfiability tests: once a hitting set

Algorithm 4 Hitting Set Tree Algorithm.

```
 1: procedure HittingSetTree(Q, K, CS, HS, w, α, p)
 2:     Input: Q (the query (a concept) to be tested for satisfiability)
 3:     Input: K (the knowledge base)
 4:     Input/Output: CS (a set of conflict sets, initially containing a single explanation)
 5:     Input/Output: HS (a set of Hitting Sets)
 6:     Input: w (the last node added to the Hitting Set Tree)
 7:     Input: E (the last axiom removed from K)
 8:     Input: p (the current edge path)
 9:     if there exists a set h ∈ HS s.t. (L(p) ∪ {E}) ⊆ h then
10:         L(w) ← X
11:         return
12:     else
13:         if Q is unsatisfiable w.r.t. K then
14:             m ←SingleMinA(Q, K)
15:             CS ← CS ∪ {m}
16:             create a new node w' and set L(w') ← m
17:             if w ≠ null then
18:                 create an edge e = ⟨w, w'⟩ with L(e) = E
19:                 p ← p ∪ e
20:             end if
21:             loop for each axiom F ∈ L(w')
22:                 HittingSetTree(A, (K − {F}), CS, HS, w', F, p)
23:             end loop
24:         else
25:             L(w) ← OK
26:             HS ← HS ∪ L(p)
27:         end if
28:     end if
29: end procedure
```

path is found any superset of that path is guaranteed to be a hitting set as well, and thus no additional unsatisfiability test are needed for that path. In this case, the algorithm labels the node with a X and makes it a leaf. 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 unsatisfiability of concept Q is represented by all distinct non leaf nodes of the tree.

Example 3 ([68])**.** *In order to describe the algorithm, let us consider a knowledge base $\mathcal{K}$ with ten axioms and an unsatisfiable concept C, corresponding to the 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 10.2 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 a hitting set tree HST in which the root node v 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 the unsatisfiability of Q w.r.t. $\mathcal{K} - \{2\}$. If it is unsatisfiable, as in our case, we obtain a new explanation for unsatisfiability of 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. removing an axiom, adding a node and checking unsatisfiability, until the unsatisfiability test turns negative, in which case we mark the new node with OK. Then, it recursively repeats these operations until the HST is fully built.

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$(Q, \mathcal{K})$

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 2 ([68])**.** *Let Q be a query and C be the unsatisfiable concept w.r.t. $\mathcal{K}$ corresponding to Q and let* ExpHST$(Q, \mathcal{K})$ *be the set of explanations returned by the hitting set algorithm, then* ExpHST$(Q, \mathcal{K})$ *is equal to the set of all explanations of unsatisfiability of the concept Q w.r.t. $\mathcal{K}$, so*

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

Proof. We divide the proof of equivalence in two parts showing that:

1. ExpHST$(Q, \mathcal{K}) \subseteq$ All-MinAs$(Q, \mathcal{K})$

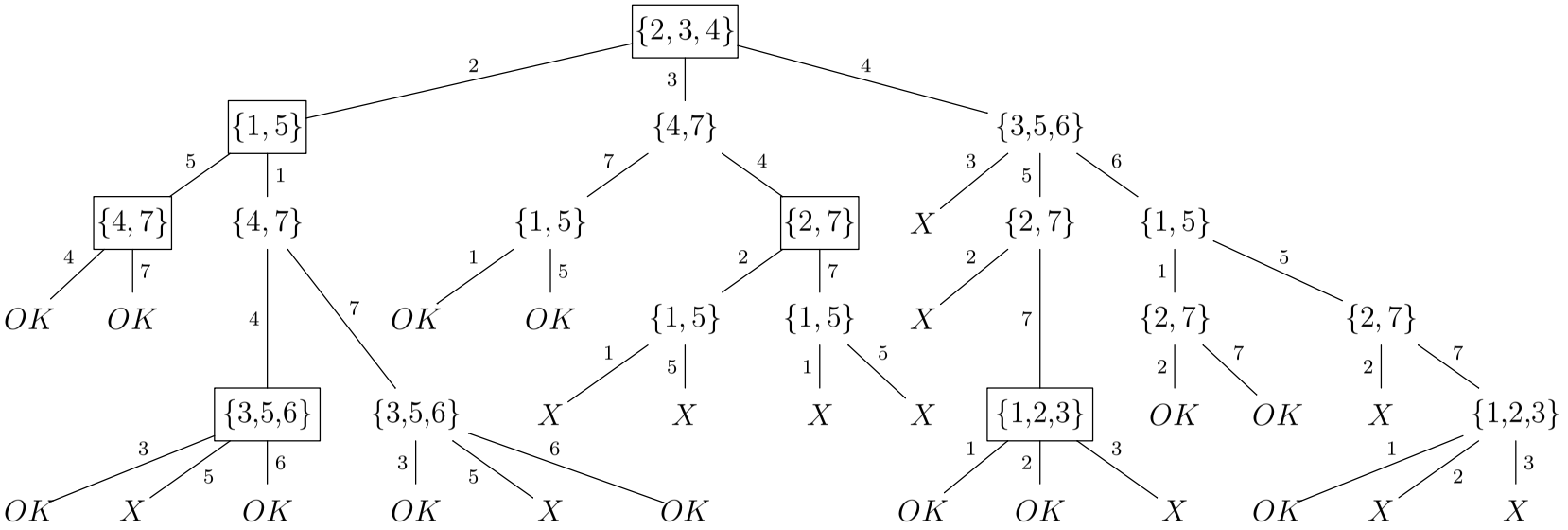


Figure 10.2: Representation of the execution of the hitting set algorithm for finding All-MinAs$(C, \mathcal{K})$. In the graph, boxed nodes are the set of distinct nodes representing a set in All-MinAs$(C, \mathcal{K})$.

2. ExpHST$(Q, \mathcal{K}) \supseteq$ All-MinAs$(Q, \mathcal{K})$

($\subseteq$ part)
Let $S \in$ ExpHST$(Q, \mathcal{K})$, then S belongs to the label of some non-leaf node w in the hitting set tree HST generated by the algorithm. In this case, $\mathcal{L}(w) \in$ All-MinAs$(Q, \mathcal{K}')$, for some $\mathcal{K}' \subseteq \mathcal{K}$. Therefore, $S \in$ All-MinAs$(Q, \mathcal{K})$.

($\supseteq$ part)
Suppose, by contradiction, there exists a set $M \in$ All-MinAs$(Q, \mathcal{K})$, but $M \notin$ ExpHST$(Q, \mathcal{K})$. In this case, M does not coincide with the label of any node in HST. Let v_0 be the root of HST, with $\mathcal{L}(v_0) = \{E_1, ..., E_n\}$, if $M = \mathcal{L}(v_0)$ then there is a contraction, otherwise there must be a $E_i \notin M$ and an edge of the graph whose label is E_i from which a path with a node labeled with M in it must be present. Since in the HST such a condition is not verified we have a contradiction. □

10.1.2 Resolving MIN-A-ENUM: Pinpointing Formula

In [5] the authors consider the problem of finding a *pinpointing formula* instead of All-MinAs$(Q, \mathcal{K})$. Thus, they propose a different extension to the tableau algorithm in which the tracing function τ associates a pinpointing formula to the labels of nodes and edges instead of explanations in the form of set of sets of axioms.

We recall here some terminology for the sake of simplicity. Each axiom E of a KB $\mathcal{K}$ is associated with a propositional variable, indicated with $var(E)$. The set of all propositional variables is indicated with $var(\mathcal{K})$.

Given a KB $\mathcal{K}$, the modified algorithm associates a monotone Boolean formula $lab(a)$ over $var(\mathcal{K})$ to every assertion a. The *insertability* of the new assertion is tested in order to decide whether a rule is applicable. Let A be a set of labeled assertions and ψ a monotone Boolean formula, the assertion a is $\psi-$*insertable* into A if either $a \notin A$, or $a \in A$ but $\psi \not\models lab(a)$. 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 $ins_\psi(B, A) := \{b \in B | b \text{ is } \psi-\text{insertable into } A\}$. For deciding the applicability of a rule we need also to give the definition of *substitution*. A substitution is a mapping $\rho : V \to D$, where V is a finite set of variables and D is a countably infinite set of constants containing all the

individuals in the KB and all the fresh individuals created by the application of the rules. Variables are seen as placeholder for individuals in the assertions. For example, an assertion can be $C(x)$ or $R(x, y)$ where C is a concept, R is a role and x and y are variables. In this case, let $C(x)$ be an assertion with the variable x and $\rho : x \to a$ a substitution, then $C(x)\rho$ denotes the assertion obtained by replacing the variable with its $\rho-image$, i.e. $C(a)$. Each expansion rule is of the form $(B_0, S) \to \{B_1, ..., B_m\}$ where B_i are finite set of assertions and S is a finite set of axioms. A rule is applicable with a substitution ρ on the variables occurring in B_0 if $S \subseteq \mathcal{K}$, $B_0\rho \subseteq A$, where A is the set of assertions contained in the ABox and found during inference, and, for every $1 \leq i \leq m$ and every substitution ρ' on the variables occurring in $B_0 \cup B_i$, we have $ins_\psi(B_i\rho', A) \neq \emptyset$, where $\psi := \bigwedge_{b \in B_0} lab(b\rho) \wedge \bigwedge_{s \in S} lab(s)$. Moreover, except for the unfold rule, the node N to which the rule is applicable is not (indirectly) blocked. When the tableau is fully built, the algorithm conjoins the labels of each clash for building the final pinpointing formula.

This approach is limited to the use of $\mathcal{ALC}$ description logics KBs as described in Section 6 of [5]. In Figure 10.1, the symbol $(*)$ denotes the rules relevant for $\mathcal{ALC}$. In these rules, the operator $\cup$ for τ means union between two sets for the approach of Section 10.1.1, while for axiom pinpointing it joins two Boolean formulas with the `OR` Boolean operator. Moreover, when a concept is already present in a node label, while the approach of the previous section checks whether to update the tracing function by verifying that the corresponding set of axioms is not a subset of τ, the pinpointing based approach performs a $\psi-insertability$ test.

Part III

A Probabilistic Semantics for Description Logics

Chapter 11

Distribution Semantics

Description Logics (DLs) are useful for modeling real world domains but they do not model uncertainty. Hence, their semantics must be extended in order to allow the definition and the management of uncertain information. To represent vague and uncertain information, DLs were extended by introducing probabilistic [58, 65, 77, 88, 99, 155] (overviewed in Chapter 13), possibilistic [61, 110] and fuzzy logics [139, 140, 142]. Our proposal is an extension of DLs, called DISPONTE for "DIstribution Semantics for Probabilistic ONTologiEs" (Spanish for "get ready"), which is based on the distribution semantics [129], a probabilistic semantics of logic programming. In the following, we focus only on probabilistic extensions because they are the most similar to our proposal. In this Chapter we give the formal definition of distribution semantics. Since it was first defined for (Probabilistic) Logic Programming, also a brief introduction on these languages together with several examples will be given in order to better understand this approach. After that, Chapter 12 presents DISPONTE while Chapter 13 discusses related works. Thus, let us start first with a discussion about some works on Probabilistic Logic Programming.

The diffusion of Logic Programming (LP) techniques made clear that an integration with probability theory was necessary to correctly model domains from the real world, where the information is often uncertain. Thus, in the early 90's many proposals [30, 97, 105, 129] have appeared that presented different probabilistic semantics for LP languages. Among them, two different approaches emerged, one that makes use of variants of the distribution semantics [129] and one that exploits Knowledge Base Model Construction

(KBMC) [152, 8].

While in the second approach a graphical model, usually a Bayesian network or a Markov network, is generated from the program for modeling the probabilistic information and computing the probability of queries, the first 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 [98], Bayesian Logic Programs [72], CLP(BN) [126] and the Prolog Factor Language [52]. These approaches specify a model through *features* that are associated with a real value, i.e. the probability value.

In Probabilistic Logic Programming (PLP), the distribution semantics underlies many languages such as Probabilistic Logic Programs [30], Probabilistic Horn Abduction [105] and its expansion Independent Choice Logic (ICL) [107], PRISM [129], pD [46], Logic Programs with Annotated Disjunctions (LPADs) [150], ProbLog [34], P-log [9] and CP-logic [149]. Despite the large number of different distribution semantics languages, 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.

As already said, all these logics exploit a semantics which defines a probability distribution over *possible worlds* and allow probabilistic choices in clauses. What differs in each logics is the definition of the distribution over logic programs.

11.1 Formal Definition

The Distribution Semantics was first presented by Sato in 1995 [129]. He presented a semantics applicable to general logic programming languages and defined the basis for probabilistic inference and learning.

Given a first order language, let F be a set of facts and R be a set of definite rules. $DB = F \cup R$ is a definite program for which the following conditions are verified: (1) DB is ground or it can be reduced to the set of all possible

Table 11.1: M_{DB_1} for the finite program $DB_1 = F_1 \cup R_1$, with $F_1 = \{A_1, A_2\}$ and $R_1 = \{B_1 \leftarrow A_1, B_1 \leftarrow A_2, B_2 \leftarrow A_2\}$.

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

ground instantiations of the clauses, (2) it is denumerably infinite and (3) it satisfies the *disjoint condition* which imposes that no atom in F unifies with the head of a rule in R.

Each ground atom A is associated to a Boolean random variable which takes value 1 if A is true and takes value 0 otherwise. An interpretation ω for F is an assignment of truth to atoms A_i in F, Ω_F is the set of all interpretations $\omega = \langle x_1, x_2, ... \rangle$, where x_i is the truth value of A_i. Now we can define a *basic distribution* P_F for F that is a probability measure on the algebra of the sample space Ω_F. The corresponding distribution function is $P_F^{(n)}(A_1, = x_1, ..., A_n = x_n)$.

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. For example, 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 $\Omega_F = \{0, 1\}_1 \times \{0, 1\}_2$ 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 11.1.

P_F is then extended to define the probability measure P_{DB} over Ω_{DB}. Ω_{DB} represents the set of all possible interpretations for ground atoms appearing in DB and $\omega \in \Omega_{DB}$ determines the truth value of every ground atom. Given a sample $\omega_{F'}$ from P_F and the set F' of atoms made true by $\omega_{F'}$, we can define the least model $M_{DB}(\omega)_{F'}$ of the definite program $F' \cup R$. $M_{DB}(\omega)_{F'}$ determines the truth value of every ground atom. Therefore, P_F can be extended to P_{DB},

thus the mass P_F on an interpretation $\omega_{F'}$ is put by P_{DB} on the least model $M_{DB}(\omega)_{F'}$. The infinite joint distribution $P_{DB} = (A_1 = x_1, ..., A_n = x_n)$ on the probabilistic ground atoms $A_1, ..., A_n$ appearing in DB for all n and $x_i \in \{0, 1\}$ identifies P_{DB_1} [130].

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

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

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

Considering the program DB_1, $\omega = \langle x_1, x_2, y_1, y_2 \rangle \in \Omega_{DB_1}$ indicates that x_i is the value of A_i and y_j is the value of B_j, where $i, j = 1, 2$. $P_{DB_1}(x_1, x_2, y_1, y_2)$ can be computed from $P_{F_1}(x_1, x_2)$.

11.2 PLP Languages under the Distribution Semantics

In the following we analyze two significant examples of distribution semantics PLP languages: LPAD and ProbLog. Before that, a description about Logic Programming, which is at the basis of PLP languages, is given.

11.2.1 Logic Programming

Work on Logic Programming (LP for short) started in the 70's, in particular, in 1974, Kowalski [79] first formalized a logic programming language. From this point, much work has followed and several extensions were presented in order to consider different type of knowledge.

One of these extensions is *disjunctive logic programming*, in which a program is a finite set of implicitly universally quantified clauses of the form

$$a_1, ..., a_n \leftarrow b_1, ...b_m \tag{11.1}$$

where $n > 0$ and $m \geq 0$. A clause can be intuitively read as "if the conjunction

of all b_js is true, then the disjunction of all a_is is true", i.e. a_1 or ... or a_n. Atoms b_j form the *body*, while atoms a_i form the *head* of the clause. The head cannot be empty, while the body can, in this case a clause is called *fact*.

In 1976, van Emden and Kowalski [147] formalized a special case of disjunctive logic programming, where a *definite logic program* requires clauses to have a single atom in the head. Formally, a definite logic program clause is a program clause of the form:

$$a \leftarrow b_1, ..., b_m \tag{11.2}$$

where $m \geq 0$, a is an atom and $b_1, ..., b_m$ are *atoms*.

In [147] the authors presented 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. Then, van Emden and Kowalski showed a procedural semantics in which one can 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 T_P operator, 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.

Prolog

Prolog was the first logic programming language and was born in 1972 at the University of Marseille. The name Prolog is the contraction of "PROgrammation en LOGique" (PROgramming in LOGic). This language is based on the ideas of Kowalski and van Emdem and was implemented by Alain Colmerauer and Philippe Roussel.

Prolog implements SLD resolution. In the following we provide an informal description of SLD resolution, please refer to [79, 147] for a formal treatment.

First, let us define some concepts useful for following the example. In clauses, atoms can contain variables, a clause is called *ground* if it does not contain variables. A *substitution* θ is an assignment of variables to some value $\theta = \{V_1/v_1, ..., V_n/v_n\}$, where V_i are variables and v_j are the values associated

with the variables. The application of a substitution to clause C (atom a), indicated with $C\theta$ ($a\theta$), is the replacement of the variables appearing in C (a) with the corresponding values as specified in θ. Given two atoms a and b and a substitution θ, if $a\theta$ and $b\theta$ are identical we say that a and b can be *unified*.

SLD resolution starts form a clause called goal or query that we want to test.

$$\leftarrow a_1, a_2, ..., a_n \ (n > 0)$$

Then, it iteratively selects a subgoal, which is an atom of the clause, and replaces this subgoal with the body of a new clause contained in the program which head *unifies* the selected subgoal. For example, if the new clause is

$$b_0 \leftarrow b_1, ..., b_m \ (m \geq 0)$$

where b_0 can be unified with a_1 through the substitution θ_1, then the goal becomes

$$\leftarrow (b_1, ..., b_m, a_2, ..., a_n)\theta_1$$

The substitution θ_1 is the *most general unifier*, i.e. there is not a more general substitution ω such that $\theta_1 = \omega\sigma$, where σ is a substitution. 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.

The results of SLD-resolution's refutation process are correct, i.e. the conclusions returned by the algorithm are logical consequences of the program, and therefore the approach is *sound*. SLD-resolution itself is also *complete*. If a goal G is a logical consequence of a program P, then there is a refutation of $P \cup \{G\}$ 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.

Example 4. *Consider the following program:*

$$sibling(X, Y) \leftarrow sibling(Y, X). \tag{11.3}$$

$$sibling(a, b). \tag{11.4}$$

with query $\leftarrow sibling(a, X)$. *The goal is unified with the head of rule (11.3) creating a new goal that is equal to the query. At this point, an infinite ap-*

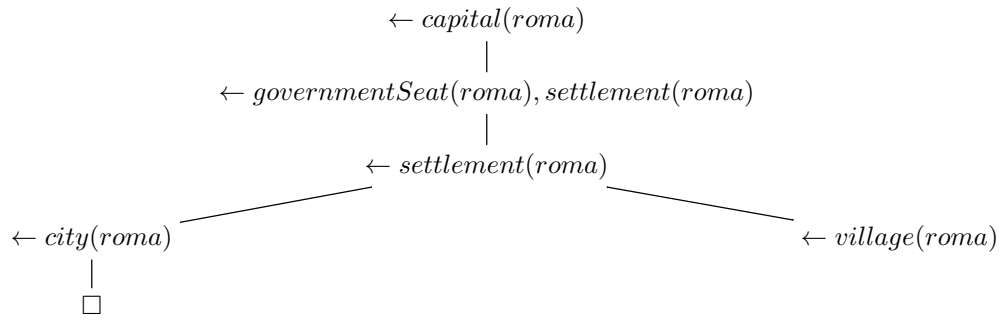


Figure 11.1: Prolog SLD resolution tree for ← *capital*(*roma*) w.r.t. the theory of Example 5.

plication of resolution will be performed. In Prolog, this inconvenience can be avoided by moving rule (11.3) textually after the fact (11.4), so that all refutations are found before going into an infinite loop.

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

Example 5. *Consider the following Prolog program*

$$capital(X) \leftarrow governmentSeat(X), settlement(X). \tag{11.5}$$
$$settlement(X) \leftarrow city(X). \tag{11.6}$$
$$settlement(X) \leftarrow village(X). \tag{11.7}$$
$$city(roma). \tag{11.8}$$
$$governmentSeat(roma). \tag{11.9}$$

For answering the query ← *capital*(*roma*)*, first the goal is rewritten using (11.5). Then, governmentSeat*(*roma*) *is removed from the goal using clause (11.9). At this point, the truth of settlement*(*roma*) *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 11.1.*

11.2.2 LPAD

A Logic Program with Annotated Disjunction (LPAD) [150] is composed of a finite set of annotated disjunctive clauses of the form

$$h_{i1} : p_{i1}; ...; h_{in_i} : p_{in_i} : -b_{i1}, ..., b_{im_i}.$$

where $p_{i1}, ..., p_{in_i}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} p_{ik} \leq 1$, $h_{i1}, ..., h_{in_i}$ are logical atoms and $b_{i1}, ..., b_{im_i}$ are logical literals. $h_{i1} : p_{i1}; ...; h_{in_i} : p_{in_i}$ is called head, while $b_{i1}, ..., b_{im_i}$ is called body. Here, it is important make two observations:

- if $n_i = 1$ and $p_{i1} = 1$, then the clause is a non-disjunctive clause;
- if $\sum_{k=1}^{n_i} p_{ik} < 1$, then the head implicitly contains an extra atom, called *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} p_{ik}$.

For obtaining a world from an LPAD we first need to ground all the different clauses, then a head atom must be chosen for each ground rule and the normal clauses with that atom only in the head form the world. In other words, for each ground clause $h_{i1}\theta_j : p_{i1}; ...; h_{in_i}\theta_j : p_{in_i} : -b_{i1}\theta_j, ..., b_{im_i}\theta_j$, where θ_j is the grounding substitution, only one $h_{ik}\theta_j$ is chosen and the clause $h_{ik}\theta_j : -b_{i1}\theta_j, ..., b_{im_i}\theta_j$ is included in the world. The probability of a world is given by the product of the probabilities associated with the selected head atoms. Finally, the probability $P(Q)$ of a query Q, i.e., a ground atom, is computed by marginalization, i.e., by summing out the worlds, as shown in following:

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

where $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise.

Example 6. *The following LPAD encodes a very simple model of the devel-*

opment of an epidemic or pandemic:

$$\begin{aligned} C_1 &= epidemic : 0.6; pandemic : 0.3 : -flu(X), cold. \\ C_2 &= cold : 0.7. \\ C_3 &= flu(david). \\ C_4 &= flu(robert). \end{aligned}$$

This LPAD models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic arises with probability of 0.6*, a pandemic arises with probability of* 0.3 *or no event (the implicit atom null) happens with probability of* 0.1*. We are uncertain about the climate since there is a probability of* 0.7 *that it is cold but we know for sure that David and Robert have the flu.*

This LPAD defines 18 possible worlds: the first rule has three atoms and two possible groundings ($X = david$ and $X = robert$), while the second rule has two head atoms, thus $3 \cdot 3 \cdot 2 = 18$*. The query is* true *only in 5 of them, therefore the probability of epidemic is* $P(epidemic) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$*. For instance, the first term is obtained from the following ground LPAD:*

$$\begin{aligned} &epidemic : 0.6 : -flu(david), cold. \\ &epidemic : 0.6 : -flu(robert), cold. \\ &cold : 0.7. \\ &flu(david). \\ &flu(robert). \end{aligned}$$

while the second from the following:

$$\begin{aligned} &epidemic : 0.6 : -flu(david), cold. \\ &pandemic : 0.3 : -flu(robert), cold. \\ &cold : 0.7. \\ &flu(david). \\ &flu(robert). \end{aligned}$$

11.2.3 ProbLog

ProbLog [34] is the language with the simplest syntax within the list presented in the introduction of this chapter.

A ProbLog program T consists of a normal logic program T_C, i.e. a finite set of clauses, and a set of independent *probabilistic facts* T_P. Each probabilistic fact is of the form

$$p_i :: F_i.$$

where p_i is a probability, i.e. $p_i \in [0, 1]$, and F_i is an atom. This means that every grounding $F_i\theta$ of F_i is *true* with probability p_i and *false* with probability $1 - p_i$.

For obtaining a world from a ProbLog program, we include in the world T_C and a subset T_P^G that contains a selection of ground probabilistic facts chosen within the set of all the grounding of the probabilistic facts in T_P. The probability of a world is the product of every p_i associated to the (ground) probabilistic facts included in the world with a factor $1-p_i$ for each grounding of a probabilistic fact not included in the world. The probability of a query Q is computed by marginalization as for LPADs.

Example 7. *The following ProbLog program T corresponds to the LPAD of Example 6:*

$$\begin{aligned}
C_1 &= epidemic : -flu(X), epid(X), cold.\\
C_2 &= pandemic : -flu(X), \backslash{+}epid(X), pand(X), cold.\\
C_3 &= flu(david).\\
C_4 &= flu(robert).\\
F_1 &= 0.7 :: cold.\\
F_2 &= 0.6 :: epid(X).\\
F_3 &= 0.3 :: pand(X).
\end{aligned}$$

In this program, $epid(X)$ and $pand(X)$ can be considered as "probabilistic activators" of the effects in the head given that the causes, i.e. $flu(X)$ and $cold$, are present. $\backslash{+}epid(X)$ means the negation of $epid(X)$.

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). 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, as in Example 6. One world where the query is true is (note that we show only the probabilistic facts):*

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

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:

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

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

11.3 Inference in Probabilistic Logic Programming

In PLP the term inference refers to the problem of computing the probability of queries. In the previous section we presented two examples of the computation of the probability of a query, where all the possible worlds are considered. In real applications this approach is unfeasible as the number of possible worlds is exponential in the number of ground probabilistic choices. An approach that was found to have good performances is that of knowledge compilation [31], where the theory and the query are compiled to a *target language* in which computing the probability has polynomial complexity. Many target languages have been proposed with different compactness and compilation algorithms, which are the computational bottleneck of the inference process.

In PLP, the inference problem can be divided in *exact inference*, *approximate inference* and *lifted inference*.

The first approach aims at computing the exact value of probability. An early attempt to exact inference compiles Relational Bayesian Networks, which extends Bayesian Networks with the possibility of defining relations between

objects, into arithmetic circuits [24]. Then the focus shifted on languages based on the distribution semantics. First the set of all the explanations for the query is found. An explanation is a minimal set of probabilistic facts that is sufficient for entailing the query. Then this set is encoded in a target language, such as Binary Decision Diagrams (BDD) or deterministic Decomposable Negation Normal Form diagrams (d-DNNF). BDDs and d-DNNFs are rooted graphs with different properties and restrictions. In particular BDDs form a subclass of d-DNNFs. Some examples of this approach are ProbLog [34] which exploits BDDs, PITA [124] which translates a general PLP program into a normal program and uses BDDs as well, and ProbLog2 [43] which converts the theory and the evidence into Boolean formulas in conjunctive normal form (CNF), i.e. conjunctions of disjunctions of literals. The Boolean formulas are then compiled to d-DNNFs. The probability is computed from BDDs and d-DNNFs by Weighted Model Counting (WMC). WMC is linear in the size of the graph.

In the approximate inference approach, the cost of the inference process is reduced by computing the probability only approximately. In one approach, a lower bound and an upper bound of the value of the probability are computed [73, 113]. A different approximate inference approach tests the truth of the query in normal programs sampled from the probabilistic program. The probability of the query is given by the fraction of the sampled programs in which the query succeeds [73]. In [119] the correspondence between LPADs and normal clauses is exploited to draw the samples, while tabling avoids sampling twice the same atom. Another possible approach is [26], where exact models are approximated by relaxing equality constraints and then improved by compensation and recovering of some equality constraints in order to reduce the approximation.

Lifted inference performs inference without first grounding the model, in order to treat indistinguishable individuals as a whole. In [11] the authors applied Lifted Variable Elimination [109] to PLP by adding two new operators to the Prolog Factor Language. A different approach for lifted inference in PLP is [146] in which a program is transformed into a d-DNNF from which the WMC is computed.

In the following we briefly describe two systems for probabilistic inference in PLP: ProbLog and PITA.

11.3.1 ProbLog Inference System

As seen in Section 11.2.3, ProbLog [34] is the simplest probabilistic extension of LP languages. A ProbLog program corresponds to a standard Prolog program in which some facts are associated with the probability that they are true. These facts are mutually independent.

In [34], the authors presented two practical approaches for Probabilistic inference, the first performs exact inference while the second approximate inference.

Both the approaches first compute the proofs of the query, then translate them into a Disjunctive Normal Form (DNF) Boolean formula and finally compute the probability of the DNF formula. The first step is executed by means of Prolog SLD-resolution. As seen in Section 11.2.1, SLD-resolution proofs can be represented using a tree where the paths from the root to the leaves are either successful or failed proofs. In particular, each path which ends to leaf whose label is the empty set represents a proof for the query. If a Boolean random variable is associated to each probabilistic fact, the probability of a single proof is the probability of the conjunctive formula built using the corresponding Boolean variables. The probability of a set of proofs is given by the probability of a DNF formula in which each conjunctive subformula represents a proof for the query.

Computing the probability of a DNF formula is a NP-hard problem. Thus knowledge compilation is exploited in order to translate the DNF formula into a BDD, form which the probability can be computed by WMC with a cost linear in the size of the graph.

ProbLog's exact approach builds a DNF formula which represents all the proofs, while the approximate approach exploits iterative deepening for building the SLD-resolution tree and two DNF formulas. These formulas can be used to compute upper and lower bounds of the probability of the query.

11.3.2 PITA

PITA [124], for "Probabilistic Inference with Tabling and Answer subsumption", is a PLP inference system also able to reason with different measures of uncertainty. PITA translates a general PLP program (an LPAD) into a normal

LP program and evaluates it by means of resolution with tabling. The tabling system is based on the use of a data structure in which each subgoal solved during the evaluation of the query is saved together with the answer of the subgoal, so that each time the subgoal is encountered again, instead of solving it, the information contained in the table is used. The tabling system is proved to be terminating and to achieve the optimal complexity for query evaluation for a wide class of programs and queries. It is used to evaluate programs with negation according to the Well-Founded Semantics. The *answer subsumption* extension of the basic tabling system is adopted in PITA in which each answer for a subgoal is combined with the others in table.

PITA is contained in `cplint`[1], a suite of programs for both inference and learning with ICL [107], LPADs [150] and CP-logic programs [149]. `cplint` can be tested online at `http://cplint.lamping.unife.it/`.

Regarding the inference problem, `cplint` includes along with PITA the following systems:

- `lpadsld`: uses a top-down procedure based on SLDNF resolution. It is able to deal with extensions of LPADs and CP-logic which allow representing CLP(BN) programs and Probabilistic Relational Models [50]

- `picl`: performs inference on ICL programs.

- `lpad`: uses a top-down procedure based on SLG resolution [25] to solve queries w.r.t. any sound LPADs, i.e., any LPAD such that each of its instances has a two-valued well founded model.

- `cpl`: uses a top-down procedure based on SLG resolution and moreover checks that the CP-logic program is valid, i.e., that it has at least an execution model.

- Modules for approximate inference which performs several different techniques, such as (dynamic) iterative deepening, k-Best, Monte Carlo [17].

[1] `https://sites.google.com/a/unife.it/ml/cplint`

11.4 Learning in Probabilistic Logic Programming

The learning problem can be described as:

Problem:

Input: Background knowledge as a probabilistic logic program B, a set of positive and negative examples E^+ and E^-, and a language bias $\mathcal{L}$.

Output: A probabilistic logic program P such that the probability of positive examples according to $P \cup B$ is maximized and the probability of negative examples is minimized.

This problem has two variants: *parameter learning* and *structure learning*. The first aims at learning the parameters of a fixed probabilistic logic program P. The examples can be given in the form of (partial) interpretations, ground atoms or (partial) proofs. Parameter learning for languages following the distribution semantics has been performed by using the Expectation Maximization (EM) algorithm or by gradient descent.

The use of combining rules in PLP under the distribution semantics leads to the presence of unobserved variables. In this case, the EM algorithm is useful given its capability to estimate the probability of models containing random variables that are not observed in the data. The EM algorithm consists of a cycle in which two steps, called *Expectation* and *Maximization*, are repeatedly performed. In the Expectation step, the distribution of the hidden variables is computed according to the current values of the parameters, while in the Maximization step the new values of the parameters are computed for the next iteration. EM is exploited in many systems such as PRISM [131], LFI-ProbLog [42], EMBLEM [13] and RIB [122]. In particular, LFI-ProbLog and EMBLEM use knowledge compilation for computing the distribution of the hidden variables, while RIB uses a special EM algorithm called information bottleneck that was shown to avoid some local maxima of EM. CEM (cplint EM) is an implementation of EM for learning parameters based on `lpadsld` reasoning module [122]. EMBLEM, CEM and RIB are included in the framework `cplint`.

Gradient descent methods compute the gradient of the target function and

iteratively modify the parameters moving in the direction of the gradient. An example of such systems is LeProbLog [55] that uses a dynamic programming algorithm for computing the gradient exploiting BDDs.

In structure learning, we are interested in inferring both the structure and the parameters of a probabilistic logic program P. An early work [78] appeared in 1997 where the authors generated an underlying graphical model using a Knowledge-Based Model Construction on which EM was applied for learning the structure of first-order rules and the associated probabilistic uncertainty parameters.

In [33] the authors presented an algorithm which, instead of adding clauses, removes as many clauses as possible from the theory in order to maximize the probability. This approach is called theory compression and works on ProbLog programs.

SEM-CP-logic [93] learns parameters and structure of ground CP-logic programs. It performs learning by considering the Bayesian networks equivalent to CP-logic programs and by applying the Structural Expectation Maximization (SEM) algorithm [45]. SEM iteratively generates refinements of the equivalent Bayesian network and it greedily chooses the one that maximizes the BIC score [135].

As seen above and in Section 11.3.2, `cplint` is an inference and learning framework which contains different systems. For solving the structure learning problem it includes the systems SLIPCASE, SLIPCOVER, CEM and LEMUR. SLIPCASE [12] iteratively refines probabilistic theories and optimizes the parameters of each theory with EMBLEM performing a beam search in the space of LPADs. SLIPCOVER [14] is an evolution of SLIPCASE that uses bottom clauses generated as in Progol [96] to guide the refinement process. In this way it is able to reduce the number of revisions thus it explore more effectively the search space. Moreover, in SLIPCOVER the space of clauses is explored with a beam search, while the space of theories is searched greedily. Both systems maximize the log likelihood of the data, which is evaluated by EMBLEM during the optimization of the parameters. The use of EMBLEM is possible because parameter learning is usually fast. LEMUR (LEarning with a Monte carlo Upgrade of tRee search) [37] is an algorithm for learning the structure of programs by searching the clause space using Monte-Carlo tree search.

Chapter 12

DISPONTE

As seen in the previous chapter, a program following the distribution semantics [129] defines a probability distribution over normal logic programs also called *worlds*. Then the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs. Sections 11.3 and 11.4 shown that many work has been done regarding inference and learning techniques under this semantics, since it is one of the most used in PLP field.

Borrowing from distribution semantics and work done in PLP, here we describe in detail DISPONTE (DIstribution Semantics for Probabilistic ONTologiEs), which assigns a meaning to probabilistic DL knowledge bases using approach similar to the distribution semantics of PLP.

In DISPONTE, a probabilistic knowledge base $\mathcal{K}$ is a set of certain axioms and/or probabilistic axioms.

- *Certain axioms* take the form of regular DL axioms.
- *Probabilistic axioms* take the form

$$p :: E \tag{12.1}$$

  where p is a real number in $[0, 1]$ and E is any DL axiom of the TBox, ABox or RBox.

The probability p of a probabilistic axiom can be interpreted as an *epistemic probability*, i.e., as the degree of our belief in axiom E. For example, a

probabilistic concept membership axiom of the form

$$p :: a : C$$

means that we have degree of belief p in $C(a)$. The statement that "Tweety flies with probability 0.9" of [57] can be expressed as

$$0.9 :: tweety : Flies$$

A *probabilistic concept inclusion axiom*

$$p :: C \sqsubseteq D \tag{12.2}$$

represents the fact that we believe in the truth of $C \sqsubseteq D$ with probability p. For example, the axiom

$$0.9 :: Bird \sqsubseteq Flies \tag{12.3}$$

means that birds fly with a 90% probability. This is different from expressing statistical information such as the degree of overlap of C and D as in Type 1 probability of [57]. Axiom (12.2) does not mean that a fraction p of individuals from C belongs to D and axiom (12.3) does not mean that 90% of birds fly.

For defining the semantics of DISPONTE we follow the approach of [108] and first give some definitions. An *atomic choice* is a couple (E_i, k) where E_i is the ith probabilistic axiom and $k \in \{0, 1\}$. k indicates whether E_i is chosen to be included in a world ($k = 1$) or not ($k = 0$). A set of atomic choices κ is *consistent* if only one decision is taken for each probabilistic axiom, i.e., $(E_i, k) \in \kappa, (E_i, m) \in \kappa \Rightarrow k = m$; we assume independence between the different choices. A *composite choice* κ is a consistent set of atomic choices. The probability of a 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.

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* in this way: $w_\sigma = \mathcal{C} \cup \{E_i | (E_i, 1) \in \sigma\}$ where $\mathcal{C}$ is the set of certain axioms. Let us indicate with $\mathcal{S}_\mathcal{K}$ the set of all selections and with $\mathcal{W}_\mathcal{K}$ the set of all worlds. The probability of a world w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(E_i,1)\in\sigma} p_i \prod_{(E_i,0)\in\sigma}(1 - p_i)$. $P(w_\sigma)$ is a probability distribution over worlds,

i.e., $\sum_{w \in \mathcal{W}_\mathcal{K}} P(w) = 1$.

A world therefore is a non-probabilistic KB that can be assigned a semantics in the usual way. A query is entailed by a world if it is true in every model of the world.

We can now assign probabilities to queries. Given a world w, the probability of a query Q is defined as $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of a query can be obtained by marginalizing the joint probability of the query and the worlds:

$$\begin{aligned} P(Q) &= \sum_{w \in \mathcal{W}_\mathcal{K}} P(Q, w) && (12.4) \\ &= \sum_{w \in \mathcal{W}_\mathcal{K}} P(Q|w)P(w) && (12.5) \\ &= \sum_{w \in \mathcal{W}_\mathcal{K} : w \models Q} P(w) && (12.6) \end{aligned}$$

where (12.4) and (12.5) follow for the sum and product rule of probability theory respectively and (12.6) holds because $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise.

Example 8. *Consider the following KB, a probabilistic version of that of Example 2 and inspired by the* `people+pets` *ontology proposed in [101]:*

$$\begin{aligned} 0.5 \quad :: \quad & \exists hasAnimal.Pet \sqsubseteq NatureLover && (12.7) \\ & \mathit{fluffy} : Cat \\ & tom : Cat \\ 0.6 \quad :: \quad & Cat \sqsubseteq Pet && (12.8) \\ & (kevin, \mathit{fluffy}) : hasAnimal \\ & (kevin, tom) : hasAnimal \end{aligned}$$

The KB indicates that the individuals that own an animal which is a pet have a 50% probability of being nature lovers and that fluffy and tom are cats. Moreover, cats have a 60% probability of being pets and that kevin has the animals fluffy and tom. The KB has four possible worlds, corresponding to the selections (each pair in the selections contains the axiom identifier and the

value of its selector k):

$$\{((12.7), 1), ((12.8), 1)\}$$
$$\{((12.7), 1), ((12.8), 0)\}$$
$$\{((12.7), 0), ((12.8), 1)\}$$
$$\{((12.7), 0), ((12.8), 0)\}$$

The query axiom $Q = kevin : NatureLover$ *is true only in the first of them, while in the remaining ones it is false, thus the probability of the query is* $P(Q) = 0.5 \cdot 0.6 = 0.3$.

Example 9. *Let us consider a slightly different knowledge base:*

$$\exists hasAnimal.Pet \sqsubseteq NatureLover$$
$$0.4 \quad :: \quad \textit{fluffy} : Cat \tag{12.9}$$
$$0.3 \quad :: \quad tom : Cat \tag{12.10}$$
$$0.6 \quad :: \quad Cat \sqsubseteq Pet \tag{12.11}$$
$$(kevin, \textit{fluffy}) : hasAnimal$$
$$(kevin, tom) : hasAnimal$$

Here individuals that own an animal which is a pet are surely nature lovers but we are not sure about the fact that fluffy and tom are cats and that cats are pets, thus we believe in this information with a certain probability. However, we know for sure that kevin has the animals fluffy and tom.

This KB has eight worlds and the query axiom $Q = kevin : NatureLover$ *is true in three of them, those corresponding to the following selections:*

$$\{((12.9), 1), ((12.10), 0), ((12.11), 1)\}$$
$$\{((12.9), 0), ((12.10), 1), ((12.11), 1)\}$$
$$\{((12.9), 1), ((12.10), 1), ((12.11), 1)\}$$

so the probability is

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

Example 10. *Consider a different KB:*

$$\begin{aligned} & kevin : \forall friend.Person \\ & (kevin, robert) : friend \\ & (robert, david) : friend \\ 0.4 \quad :: \quad & Trans(friend) \end{aligned} \tag{12.12}$$

Here we know that all individuals in the friend relationship with kevin are persons. Moreover, kevin is a friend of robert, that is in turn a friend of david and that, given three individuals a, b and c, there is a 40% probability that if a is a friend of b and b is a friend of c then a is a friend of c. In this particular case, this means that we have a 40% probability that, if kevin is a friend of robert and robert is a friend of david, then kevin is a friend of david. Since the first two are certain facts, then kevin is a friend of david with a 40% probability and david is a person also with a 40% probability. This KB has two worlds, one which does not contain axiom (12.12) and one that contains it. Both the two queries are true in only the world that contains axiom (12.12).

In DISPONTE, apparently contradictory probabilistic information is allowed. For example, the KB

$$\begin{aligned} 0.9 \quad :: \quad & Bird \sqsubseteq Flies \\ 0.1 \quad :: \quad & tweety : Flies \\ & tweety : Bird \end{aligned}$$

states that the probability of flying for a bird is 0.9 and the probability of flying for *tweety*, a particular bird, is 0.1. The two probabilistic statements are considered as independent evidence for *tweety* flying and are combined giving the probability 0.91 for the query $tweety : Flies$. In fact, this KB has four worlds and $tweety : Flies$ is true in three of them, giving $P(Q) = 0.9 \cdot 0.1 + 0.9 \cdot 0.9 + 0.1 \cdot 0.1 = 0.91$. Thus knowledge about instances of the domain may reinforce general knowledge and vice-versa.

However, note that if the regular DL KB obtained by stripping the probabilistic annotations is inconsistent, then there will also be worlds that are inconsistent. These worlds will entail the query trivially, as does the regular KB.

An inconsistent DISPONTE KB should not be used to derive consequences, just as a regular inconsistent DL KB shouldn't.

In 2007, W3C formed the Uncertainty Reasoning for the World Wide Web Incubator Group (URW3-XG) in order to specify requirements for managing uncertain information in the World Wide Web. In 2008, the URW3-XG produced a final report [144] where it discusses the challenges of reasoning with uncertain information on the World Wide Web by highlighting several use cases for the representation of uncertainty:

- 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 healthcare and life sciences information and knowledge.

DISPONTE is a candidate formalism for tackling these problems since it introduces probability in general expressive description logics such as $\mathcal{SROIQ}(\mathbf{D})$ that is one of the bases of OWL. In the following example we show how DISPONTE can handle information coming from different untrusted sources.

Example 11. *Consider a KB similar to the one of Example 9. Suppose the user has the knowledge*

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

In this example we consider a single cat, fluffy, and are two sources of information with different reliability that provide the information that fluffy is a cat. On one source the user has a degree of belief of 0.4, i.e., he thinks it is correct with a 40% probability, while on the other source he has a degree of

belief 0.3, i.e., he thinks it is correct with a 30% probability. The user can reason on this knowledge by adding the following statements to his KB:

$$0.4 \quad :: \quad \textit{fluffy} : Cat \tag{12.13}$$

$$0.3 \quad :: \quad \textit{fluffy} : Cat \tag{12.14}$$

The two statements represent independent evidence on fluffy being a cat. The query axiom $Q = kevin : NatureLover$ is true in three out of the four worlds, those corresponding to the selections:

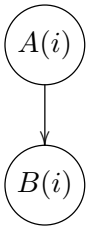
$$\{((12.13), 1), ((12.14), 1)\}$$
$$\{((12.13), 1), ((12.14), 0)\}$$
$$\{((12.13), 0), ((12.14), 1)\}$$

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, the probability comes from the disjunction of two independent Boolean random variables with probabilities respectively 0.4 and 0.3:

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

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 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 12.1.

The joint probability distribution $P(A(i), B(i))$ over the two Boolean ran-



Prob. Table 1: $P(A(i))$

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

Prob. Table 2: $P(B(i)|A(i))$

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

Figure 12.1: Bayesian Network representing the dependency between $A(i)$ and $B(i)$.

dom 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:

$$p_1 :: i : A \tag{12.15}$$

$$p_2 :: \neg A \sqsubseteq B \tag{12.16}$$

$$p_3 :: A \sqsubseteq B \tag{12.17}$$

We can associate the Boolean random variables X_1 with (12.15), X_2 with (12.16) and X_3 with (12.17), 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

$$w_1 = \{\},\ w_2 = \{(12.17)\}$$

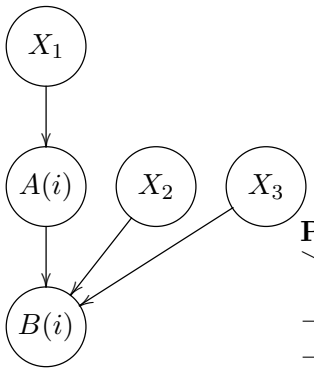
Let us call P' the probability distribution defined by the above KB. Then the probabilities of w_1 and w_2 are

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

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 12.1 with the Bayesian network of Figure 12.2.



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

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 12.2 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.

Chapter 13

Probabilistic Description Logics

A classification between different types of first-order logics of probability was first presented by Bacchus [7] and then by Halpern [57]. It defines different types of probability:

Type 1 allows the definition of *statistical* information about the world. It puts probability on entities of the domain and permits the definition of statements such as "The probability of a randomly chosen individual in the domain which is a bird flies with a probability of 0.9". It means that the 90% of the birds in a population have the property of flying.

Type 2 allows the definition of *epistemic* information which defines a *degree of belief*. It puts probability on possible worlds where statements such as "The probability that Tweety flies is 0.9" can be asserted.

In [57] the authors proposed a probabilistic extension of OWL for combining the two types of probability in one framework where statements such as "The probabilities that Tweety flies is greater than the probability that a randomly chosen bird flies" can be expressed. DISPONTE allows defining only "Type 2" statements since the probability associated with an axiom represents the degree of belief in that axiom as a whole.

Prob-$\mathcal{ALC}$ [92] derives directly from Halpern's probabilistic first order logic and considers only "Type 2" 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 while DISPONTE allows probabilistic axioms.

Heinsohn [58] extended the DL $\mathcal{ALC}$ in order to allow the definition of statistical information of the form $P(C|D) = [p_l, p_u]$ called *probabilistic terminological axioms*, where C, D are concept descriptions and $0 \leq p_l \leq p_u \leq 1$ are real numbers. It states that the conditional probability for an object belonging to D of belonging also to C is in the interval $[p_l, p_u]$. The formal semantics of the extended language is defined in terms of probability measures on the set of all concept descriptions. Given a finite interpretation $\mathcal{I}$, it satisfies $P(C|D) = [p_l, p_u]$ iff

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

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, such that in all the probability measures satisfying $\mathcal{K}$ the conditional probability $P(C|D)$ belongs to the interval $[p, q]$, denoted by $\mathcal{K} \models P(C|D) \in [p, q]$. Thus, the aim of inference is to derive optimal bounds for additional conditional probabilities. Heinsohn introduced local inference rules for deriving bounds but its approach is not complete, hence the rules are not sufficient to derive optimal bounds. Heinsohn does not allow probabilistic assertional knowledge about concept and role instances.

Jaeger [65] extended [58] by allowing the definition of probabilistic assertion of the form $P(C(a)) = p$, where C is a concept, a is an individual and $p \in [0, 1]$ a real number. The definition of the formal semantics makes use of probability measures on the set of all concept descriptions, one for each individual name. Given a knowledge base $\mathcal{K}$, the inference problem is to derive optimal bounds for additional probabilistic assertions, thus the approach presented by Heinsohn must be extended to take into account the connection between the probability measures of the terminological part and those of the assertional part. Jaeger describes two approaches, a naive method for computing optimal bound using cross-entropy minimization and one that reduces

the inference problem to a linear optimization problem. Both approaches do not allow "Type 2" statements as DISPONTE.

Moreover, DISPONTE differs from [58, 65] because it provides a unified framework for representing different types of probabilistic knowledge: from assertional to terminological degree of belief knowledge.

PR-OWL [21, 28] is a probabilistic extension for OWL that consists of a set of classes, subclasses and properties that collectively form a framework for building probabilistic ontologies. It allows the use of the first-order probabilistic logic MEBN [81] for representing uncertainty in ontologies. DISPONTE differs from [21, 28] because it does not resort to a full-fledged first-order probabilistic language, allowing the reuse of inference technology from DLs.

A different approach to the combination of DLs with probability is taken in [51, 87, 88] where authors presented P-$\mathcal{SHIQ}(\mathbf{D})$, a DL probabilistic semantics that exploits probabilistic lexicographic entailment from probabilistic default reasoning. P-$\mathcal{SHIQ}(\mathbf{D})$ allows both terminological probabilistic knowledge as well as assertional probabilistic knowledge about instances of concepts and roles. Terminological probabilistic knowledge is expressed using *conditional constraints* of the form $(D|C)[l,u]$ as in [58] 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 contraints 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 $[l,u]$. Similarly to [65], the terminological knowledge is interpreted statistically while the assertional knowledge is interpreted in an epistemic way by assigning degrees of beliefs to assertions. Therefore, while assertional probabilistic knowledge in P-$\mathcal{SHIQ}(\mathbf{D})$ corresponds to that of DISPONTE, terminological probabilistic knowledge refers to Type 1 probabilistic information. Moreover P-$\mathcal{SHIQ}(\mathbf{D})$ also allows expressing default knowledge about concepts that can be overridden in subconcepts and whose semantics is given by Lehmann's lexicographic default entailment [82]. These works are based on Nilsson's probabilistic logic [100], where a probabilistic interpretation Pr defines a probability distribution over the set of interpretations Int, i.e., $\sum_{Int} Pr(Int) = 1$. The probability of a logical formula F according to Pr, denoted $Pr(F)$, is the sum of all $Pr(I)$

such that $I \in Int$ and $I \models F$. A probabilistic knowledge base $\mathcal{K}$ is a set of probabilistic formulas of the form $F \geq p$. A probabilistic interpretation Pr satisfies $F \geq p$ iff $Pr(F) \geq p$. Pr satisfies $\mathcal{K}$, or Pr is a model of $\mathcal{K}$, iff Pr satisfies all $F \geq p \in \mathcal{K}$. $Pr(F) \geq p$ is a tight logical consequence of $\mathcal{K}$ iff p is the infimum of $Pr(F)$ in the set of all models Pr of $\mathcal{K}$. Tight logical consequences from probabilistic knowledge bases can be computed by solving a linear optimization problem.

Nilsson's probabilistic logic considers sets of distributions, differently, the distribution semantics computes a single distribution over possible worlds, thus they allow different conclusions. Consider for example a probabilistic theory composed of $C(a) \geq 0.4$ and $C(b) \geq 0.5$ and a DISPONTE KB composed of the axioms $0.4 :: a : C$ and $0.5 :: b : C$; with Nilsson's probabilistic logic allows to conclude the lowest p such that $Pr(C(a) \vee C(b)) \geq p$ holds is 0.5, while DISPONTE allows to state that $P(a : C \vee b : C) = 0.7$. This is due to the fact that differently from the Nilsson's logic, where no assumption about the independence of the statements is made, in the distribution semantics the probabilistic axioms are considered as independent. While independencies can be encoded in Nilsson's logic by carefully choosing the values of the parameters, reading off the independencies from the theories becomes more difficult.

The work reported in [67] presents an approach for computing answer probabilities to conjunctive queries w.r.t. probabilistic databases in the presence of an OWL2 QL ontology. Each assertion is assumed to be stored in a database and associated with probabilistic events. Probabilities can occur only in the data, but neither in the ontology nor in the query and all atomic events are assumed to be probabilistically independent, resulting in a semantics very similar to the distribution semantics. The authors proposed two distinct types of ABoxes where events can be seen as Boolean combinations of atomic events or only as atomic events. For Boolean conjunctive queries, the latter setting is subsumed by DISPONTE. Only very simple conjunctive queries in the latter setting can be answered in PTime, while most queries are #P-hard. The authors underline the general interest and usefulness of the approach for a wide range of applications including the management of data extracted from the web, machine translation, and dealing with data that arise from sensor networks.

$\mathcal{EL}^{++}$-LL [99] presents the combination of DLs $\mathcal{EL}^{++}$ with probabilistic log-linear models. Every axiom can be probabilistic, in this case the axiom is associated with a real-valued weight which defines a degree of confidence (Type 2). The higher the value, the higher the confidence of its truth. The semantics is defined by a log-linear probability distribution over consistent ontologies, i.e., a joint distribution over consistent sets of concept and role inclusion axioms. The probabilistic KB is mapped to a set of (instantiated) first-order formulas. After the mapping, the KB is queried by transforming the inference problem into an integer linear program, thus inference is seen as an optimization problem. This approach differs from DISPONTE mainly in the probabilistic distribution.

Other approaches exploit the use of graphical models. In [38] the authors proposed a probabilistic extension of OWL that admits a translation 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 [38] because it specifies a distribution over worlds rather than individuals.

Koeller et al. [77] 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. They exploit Bayesian networks to specify the distribution because the set of descriptions is infinite, therefore a way to define a finite specification is needed. The probability $P(C)$ of a concept description can be computed by means of a Bayesian network inference algorithm. DISPONTE differs from this approach because it allows representing different types of probabilistic knowledge in a unified way. A different combination of Bayesian network with DLs is presented in [155] where, instead of extending DLs, the focus is on the extension of Bayesian networks. In this way, the restrictions on the network imposed by [77] are avoided.

Another extension of $\mathcal{ALC}$ is CR$\mathcal{ALC}$ [91]. It adopts a semantics based on interpretations and, differently form DISPONTE, allows the expression of both probability types. Type 1 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 $G(\mathcal{K})$ in which a node represents a concept, a role or a restriction $\exists R.C$ and $\forall R.C$ and the edges represent the relations between them: (1) if a concept C directly uses concept D, then D is a parent of C in $G(\mathcal{K})$, (2) if a node is a restriction, $G(\mathcal{K})$ contains an edge from R to each restriction directly using it and from each restriction to the concept C appearing in it. $G(\mathcal{K})$ is then used together with the domain for generating a ground graph in which each node represents an instantiated logical atom $C(a)$ or $R(a, b)$. Inference is performed by means of a first order loopy belief propagation algorithm on the ground graph.

In [29], *DL-Lite* is combined with Bayesian networks while in [53] Datalog$^{\pm}$ is combined with Markov networks such that a KB is composed of a set of annotated axioms and a graphical model. 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 computed by summing out the probabilities of the possible worlds where the query holds. DISPONTE represents a special case of [29, 53] in which each probabilistic axiom is annotated with a single random variable and the graphical model encodes independence among the random variables. This special case is of interest as inference technology from DLs can be directly employed.

Similar to [29], in [22] 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 annotated 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.

An even different approach is given by the combination between DLs and logic programs. In [20], 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. P may contain probabilistic alternatives in the style of ICL [107]. Interpretations assign a probability to ground atoms, in the style of Nilsson probabilistic logic [100]. Queries can be answered by finding all answer sets. Differently from [20], in DISPONTE interpretations are not probabilistic and they are assigned a probability, instead of being a mapping from atoms to probabilities.

Part IV

Inference in Probabilistic DLs

Chapter 14

Inference

In Chapter 12 we have seen that for computing probability of queries we can list the possible worlds in which the query is true. Generally this is not feasible since the number of worlds is exponential in the number of probabilistic axioms. We propose an approach for performing inference over DISPONTE DLs in which we first find explanations for the given query and then compute the probability of the query from them. In the following we discuss the approach and we present three algorithms, named BUNDLE, TRILL and TRILL^P, which are able to execute probabilistic inference, thus they are able to find explanations for queries and compute their probabilities. The three systems are presented in chapters 15, 16 and 17 respectively. Then, Chapter 18 discusses the complexity of the approach presented here, while Chapter 19 describes related works. Finally, Chapter 20 shows the experiments done to test our algorithms.

In order to discuss the approach for probabilistic inference, we recall some useful definitions.

A composite choice κ *identifies* a set of worlds $\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_\mathcal{K}, \sigma \supseteq \kappa\}$, the set of worlds whose selection is a superset of κ, i.e., the set of worlds "compatible" with κ. We define the set of worlds *identified* by a set of composite choices K as $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$.

A composite choice κ is an *explanation* for a query Q if Q is entailed by every world of ω_κ. A set of explanations, corresponding to a set of composite choices K, is *covering* with respect to Q if every world $w_\sigma \in \mathcal{W}_\mathcal{K}$ in which Q is entailed is included in ω_K.

Two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent. For example, given a probabilistic axiom E_i, the composite choices $\kappa_1 = \{(E_i, 1)\}$ 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 κ_1 and κ_2 are incompatible. For example

$$K = \{\kappa_1, \kappa_2\} \tag{14.1}$$

with

$$\kappa_1 = \{(E_i, 1)\}$$

and

$$\kappa_2 = \{(E_i, 0), (E_l, 1)\} \tag{14.2}$$

is pairwise incompatible.

We define *the probability of a pairwise incompatible set of composite choices* K as

$$P(K) = \sum_{\kappa \in K} P(\kappa) \tag{14.3}$$

Two sets of composite choices K_1 and K_2 are *equivalent* if $\omega_{K_1} = \omega_{K_2}$, i.e., if they identify the same set of worlds. For example, K in (14.1) is equivalent to

$$K' = \{\kappa'_1, \kappa'_2\} \tag{14.4}$$

with

$$\kappa'_1 = \{(E_i, 1)\}$$

and

$$\kappa'_2 = \{(E_l, 1)\} \tag{14.5}$$

14.1 Splitting Algorithm

Let κ be a composite choice and E be an axiom such that $\kappa \cap \{(E, 0), (E, 1)\} = \emptyset$, the *split* of κ on E is the set of composite choices $S_{\kappa,E} = \{\kappa \cup \{(E, 0)\}, \kappa \cup \{(E, 1)\}\}$. It is easy to see that κ and $S_{\kappa,E}$ identify the same set of possible worlds, i.e., that $\omega_\kappa = \omega_{S_{\kappa,E}}$. For example, the split of κ'_2 in (14.5) on E_i contains κ_2 in (14.2) and $\{(E_i, 1), (E_l, 1)\}$.

Theorem 3 ([108]). *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 $(E, k) \in \kappa_1 \setminus \kappa_2$. We replace κ_2 by the split of κ_2 on E. Let $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2, E}$.

In both cases $\omega_K = \omega_{K'}$. A splitting algorithm, shown in Algorithm 5, repeatedly executes these two operations until no one can be applied. Since K is a finite set of finite composite choices, the algorithm terminates. The resulting set K' is pairwise incompatible and is equivalent to the original set. For example, the splitting algorithm applied to K' in (14.4) can results in K in (14.1). □

Algorithm 5 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 (E, k) ∈ κ1 \ κ2
10:                K ← K \ {κ2} ∪ S_{κ2,E}
11:            else
12:                exit and return K
13:            end if
14:        end if
15:    end loop
16: end procedure
```

Theorem 4 (Lemma A.8, [106]). *If K_1 and K_2 are both mutually incompatible finite sets of finite composite choices such that they are equivalent, then $P(K_1) = P(K_2)$.*

Proof. Let D be the set of all axioms appearing in an atomic choice in either K_1 or K_2. The set D has the following properties: (1) is finite and (2) each composite choice in K_1 and K_2 has atomic choices for a subset of D. For each composite choice κ of both K_1 and K_2, we repeatedly replace κ with its split $S_{\kappa,E}$ on an axiom E from D that does not appear in κ. This procedure does not change the total probability since the sum of probabilities of $(E, 0)$ and $(E, 1)$ is 1. At the end of this procedure the two sets of composite choices will be identical. In fact, any difference can be extended into a possible world belonging to ω_{K_1} but not to ω_{K_2} or vice versa, contradicting the hypothesis. □

For example, K in (14.1) and $K'' = \{\kappa''_1, \kappa''_2\}$ with $\kappa''_1 = \{(E_i, 1), (E_l, 0)\}$ and $\kappa''_2 = \{(E_l, 1)\}$ are equivalent and are both pairwise incompatible. Their probabilities are

$$P(K) = p_i + (1 - p_i) \cdot p_l = p_i + p_l - p_i \cdot p_l$$

and

$$P(K'') = p_i \cdot (1 - p_l) + p_l = p_i + p_l - p_i \cdot p_l$$

Note that if we compute the probability of K' in (14.4) with formula (14.3) we would obtain $p_i + p_l$ which is different from the probabilities of K and K'' above, even if K' is equivalent to K and K''. This is due to the fact that K' is not pairwise incompatible. Thus the probability $P(K)$ of a generic set of composite choices K is $P(K) = P(K')$, where K' is a mutually incompatible set of composite choices that is equivalent to K, i.e., such that $\omega_{K'} = \omega_K$. Given a query Q, the set $K_Q = \{\sigma | \sigma \in \mathcal{S}_\mathcal{K} \wedge w_\sigma \models Q\}$ is a set of pairwise incompatible composite choices. Since $P(Q) = \sum_{\sigma \in K_Q} P(\sigma)$, then $P(Q) = P(K_Q)$. If K' is a covering set of explanations for Q, then K' and K_Q are equivalent so $P(Q) = P(K_Q) = P(K')$. This proves that in order to compute the probability of a query we do not have to generate all worlds where a query is true, instead finding a mutually incompatible covering set of explanations is enough. Moreover, an additional result is given by the following theorem.

Theorem 5. *Given two finite sets of finite composite choices K_1 and K_2, if $K_1 \subseteq K_2$, then $P(K_1) \leq P(K_2)$.*

Proof. Let K'_1 and K'_2 be the result of the application of the splitting algorithm

to K_1 and K_2 respectively. During the application of the splitting algorithm to K_2 we obtain, in an intermediate step, $K = K'_1 \cup K'$ for a certain K'. In the continuation of the algorithm the steps below will be executed:

- If there exists a $\kappa \in K$ and a $\kappa' \in K$ such that $\kappa' \subseteq \kappa$, at least one of κ and κ' must not belong to K'_1, otherwise κ would have been removed when splitting K'_1. In this case, κ is removed from K while κ' remains.
- If there is a compatible couple κ_1 and κ_2 in K, we can assume that one of the two, say κ_2, does not belong to K'_1, since otherwise it would have been split before. We add to K the split of κ_2 on an atomic choice in κ_1 but not in κ_2.

At the end of the execution, for each element κ_1 of K'_1, K'_2 contains an element κ_2 such that $\kappa_2 \subseteq \kappa_1$. Therefore, in the summation in (14.3), for each term in $P(K'_1)$ there will be a term in $P(K'_2)$ with a larger or equal value so $P(K'_1) \leq P(K'_2)$. □

Theorem 5 implies that if K is a finite set of finite explanations for a query Q that is not covering, i.e., K does not contain all possible explanations for Q, then $P(K)$ will be a lower bound of $P(Q)$. Starting from this lower bound and considering an increasing set of explanations, we can compute progressively more accurate estimates of $P(Q)$. When K contains all possible explanations, then $P(K) = P(Q)$.

The problem of computing the probability of a query can thus be reduced to finding a covering set of explanations K and then making it pairwise incompatible, so that the probability can be computed with the summation of (14.3). To obtain a mutually incompatible set of explanations, the splitting algorithm can be applied.

Example 12. *Let us consider the KBs of Example 8 and Example 10. Regarding the first, for the query* $Q = kevin : NatureLover$ *there is only one explanation, thus the covering set if* $K = \{\{((12.7), 1), ((12.8), 1)\}\}$. *Hence we can directly compute the probability of the query* $P(Q) = 0.5 \cdot 0.6 = 0.3$ *which corresponds to the probability given by the semantics.*

Analogously, in Example 10 for the query $Q_1 = (kevin, david) : friend$ *and the query* $Q_2 = david : Person$ *the covering sets of explanations are equal for*

both queries and contain only one explanation, thus the probability of the two queries is the same, i.e., $P(Q_1) = P(Q_2) = 0.4$.

Example 13. *Let us now consider the KB of Example 9. A covering set of explanations for the query axiom* $Q = kevin : NatureLover$ *is* $K = \{\kappa_1, \kappa_2\}$ *where* $\kappa_1 = \{((12.9), 1), ((12.11), 1)\}$ *and* $\kappa_2 = \{((12.10), 1), ((12.11), 1)\}$.

In this example K *contains more than one explanation thus splitting algorithm must be executed to make explanations mutually incompatible. An equivalent pairwise incompatible set* K' *of explanations is* $K' = \{\kappa'_1, \kappa'_2\}$ *where*

$$\kappa'_1 = \{((12.9), 1), ((12.11), 1), ((12.10), 0)\}$$
$$\kappa'_2 = \{((12.10), 1), ((12.11), 1)\}$$

So $P(Q) = 0.4 \cdot 0.6 \cdot 0.7 + 0.3 \cdot 0.6 = 0.348$ *which corresponds to the probability given by the semantics.*

14.2 Binary Decision Diagrams

Given a covering set of explanations K (not necessarily mutually incompatible) for a query Q, we can define the Disjunctive Normal Form (DNF) Boolean formula f_K as

$$f_K(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(E_i,1)\in\kappa} X_i \bigwedge_{(E_i,0)\in\kappa} \overline{X_i} \tag{14.6}$$

The variables $\mathbf{X} = \{X_i | (E_i, k) \in \kappa, \kappa \in K\}$ are independent Boolean random variables. The probability of being true for variable X_i is p_i, where p_i is the probability associated with axiom E_i. The probability that $f_K(\mathbf{X})$ assumes value 1 is equal to the probability of Q. Following the approaches seen in Section 11.3, we can now apply *knowledge compilation* to the propositional formula $f_K(\mathbf{X})$ [31] in order to translate it into a target language that allows answering queries in polynomial time. Many proposals of target languages have been presented, we use here Binary Decision Diagrams (BDDs) that was found to give good performances. From a BDD we can compute the probability of the query with weighted model counting, performed by a dynamic programming algorithm that is linear in the size of the BDD [34]. This method performs better than the splitting algorithm in practice [73, 118].

A BDD is a rooted, directed acyclic graph in which

- there is one level for each Boolean variable,
- each node n in a BDD is associated with a Boolean variable
- each node n has two outgoing edges: one corresponding to the 1 value of the variable associated with the level of n, indicated with $child_1(n)$, and one corresponding to the 0 value of the variable, indicated with $child_0(n)$. When drawing BDDs, the 0-branch - the one going to $child_0(n)$ - is distinguished from the 1-branch by drawing it with a dashed line.
- the leaves store either 0 or 1

BDDs can be built by combining simpler BDDs using Boolean operators. While building BDDs, simplification operations can be applied that delete or merge nodes. Merging is performed when the diagram contains two identical sub-diagrams, while deletion is performed when both arcs from a node point to the same node. In this way a reduced BDD is obtained, often with a much smaller number of nodes with respect to the original BDD. The size of the reduced BDD highly depends on the order of the variables. A BDD is ordered if in all paths through the graph the variables respect a given linear order $X_1 < X_2 < ... < X_n$, i.e., if we take two paths there are not two or more variables in a different order. Finding an optimal order is an NP-complete problem [16] and several heuristic techniques are used in practice by highly efficient software packages such as CUDD[1]. Alternative methods involve learning variable orders from examples [54].

Since BDDs represents Boolean formulas, given an assignment for all the variables, a BDD can be used to compute the value of the formula by traversing the graph starting from the root to a leaf. The traversal is done on the basis of the value of the variable corresponding to the node. When a leaf is reached the value associated is returned. For instance, a BDD for the function

$$f(\mathbf{X}) = (X_1 \wedge X_3) \vee (X_2 \wedge X_3) \tag{14.7}$$

is shown in Figure 14.1.

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

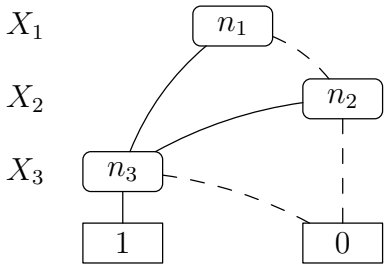


Figure 14.1: BDD for function (14.7).

A BDD performs a Shannon expansion of the Boolean formula $f_K(\mathbf{X})$, so that, if X is the variable associated with the root level of a BDD, the formula $f_K(\mathbf{X})$ can be represented as $f_K(\mathbf{X}) = X \wedge f_K^X(\mathbf{X}) \vee \overline{X} \wedge f_K^{\overline{X}}(\mathbf{X})$ where $f_K^X(\mathbf{X})$ ($f_K^{\overline{X}}(\mathbf{X})$) is the formula obtained by $f_K(\mathbf{X})$ by setting X to 1 (0). Now the two disjuncts are pairwise exclusive and the probability of $f_K(\mathbf{X})$ can be computed as $P(f_K(\mathbf{X})) = P(X) \cdot P(f_K^X(\mathbf{X})) + (1 - P(X)) \cdot P(f_K^{\overline{X}}(\mathbf{X}))$. In this way, BDDs make the explanations pairwise incompatible. Given the BDD, we can use the function PROB shown in Algorithm 6 for performing weighted model counting. This dynamic programming algorithm traverses the diagram from the leaves and computes the probability of a formula encoded as a BDD.

The function stores the value of already visited nodes in a table so that, if a node is visited again, its probability can be retrieved from the table. This optimization is fundamental to achieve linear cost in the number of nodes, as without it the cost of the function PROB would be proportional to 2^n where n is the number of Boolean variables.

Let us discuss inference on some examples.

Example 14 (Example 13 cont.)**.** *Instead of the splitting algorithm, here we use BDDs to compute $P(Q)$. We first recall the covering set $K = \{\kappa_1, \kappa_2\}$ of explanations for the query axiom*

$$\kappa_1 = \{((12.9), 1), ((12.11), 1)\}$$
$$\kappa_2 = \{((12.10), 1), ((12.11), 1)\}$$

If we associate the random variables X_1 to (12.9), X_2 to (12.10) and X_3 to (12.11), $f_K(\mathbf{X})$ is shown in (14.7) and the BDD associated with the set K of explanations is shown in Figure 14.1. By applying the function PROB *in*

Algorithm 6 Function PROB: it takes a BDD encoding a formula and computes its probability.

```
 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
```

Algorithm 6 to this BDD we get

$$\begin{aligned} \text{PROB}(n_3) &= 0.6 \cdot 1 + 0.4 \cdot 0 = 0.6 \\ \text{PROB}(n_2) &= 0.4 \cdot 0.6 + 0.6 \cdot 0 = 0.24 \\ \text{PROB}(n_1) &= 0.3 \cdot 0.6 + 0.7 \cdot 0.24 = 0.348 \end{aligned}$$

and therefore $P(Q) = \text{PROB}(n_1) = 0.348$*, which corresponds to the probability given by the semantics.*

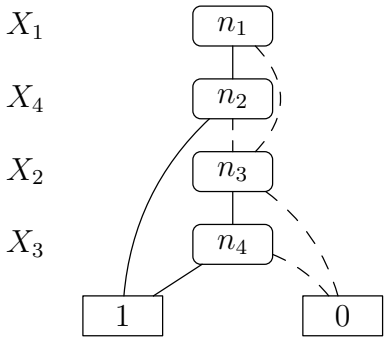


Figure 14.2: BDD for Example 15.

Example 15. *Let us now consider a slightly different knowledge base:*

$$\exists hasAnimal.Pet \sqsubseteq NatureLover$$
$$(kevin, fluffy) : hasAnimal$$
$$(kevin, tom) : hasAnimal$$
$$0.4 \quad :: \quad fluffy : Dog \tag{14.8}$$
$$0.3 \quad :: \quad tom : Cat \tag{14.9}$$
$$0.6 \quad :: \quad Cat \sqsubseteq Pet \tag{14.10}$$
$$0.5 \quad :: \quad Dog \sqsubseteq Pet \tag{14.11}$$

A covering set of explanations for the query axiom $Q = kevin : NatureLover$ is $K = \{\kappa_1, \kappa_2\}$ where $\kappa_1 = \{((14.8), 1), ((14.11), 1)\}$ and $\kappa_2 = \{((14.9), 1), ((14.10), 1)\}$. If we associate the random variables X_1 to (14.8), X_2 to (14.9), X_3 to (14.10) and X_4 to (14.11), the BDD associated with the set K of explanations is shown in Figure 14.2.
By applying the function Prob *in Algorithm 6 we get*

$$\begin{aligned}
\textsc{Prob}(n_4) &= 0.6 \cdot 1 + 0.4 \cdot 0 = 0.6 \\
\textsc{Prob}(n_3) &= 0.3 \cdot 0.6 + 0.7 \cdot 0 = 0.18 \\
\textsc{Prob}(n_2) &= 0.5 \cdot 1 + 0.5 \cdot 0.18 = 0.59 \\
\textsc{Prob}(n_1) &= 0.4 \cdot 0.59 + 0.6 \cdot 0.18 = 0.344
\end{aligned}$$

so $P(Q) = \textsc{Prob}(n_1) = 0.344$.

Chapter 15

BUNDLE

The problem of finding explanations for a query has been investigated in Chapter 10 where we called it *axiom pinpointing* and we defined it as the problem MIN-A-ENUM which concern the computation of the set of all possible MinAs ALL-MINAS$(Q, \mathcal{K})$, i.e., the set of all possible explanations. ALL-MINAS$(Q, \mathcal{K})$ is a covering set of explanations.

In Section 10.1.1 we described how the Pellet reasoner [136] solves MIN-A-ENUM. It implements a *hitting set tree* algorithm which repeatedly calls a modified tableau algorithm that builds a MinA from a KB from which some axioms are removed depending on the previously found explanations.

BUNDLE is based on Pellet and uses it for solving the MIN-A-ENUM problem. BUNDLE algorithm computes the probability of queries from a probabilistic knowledge base that follows DISPONTE by first finding a covering set of explanations for the query and then making the explanations pairwise incompatible by using BDDs. Finally, it computes the probability from the BDD by using function PROB of Algorithm 6.

The main procedure, shown in Algorithm 7, first builds a data structure $PMap$ that associates each probabilistic DL axiom E_i with its probability p_i [line 8]. In case E_i is associated with more than a probability value, BUNDLE first aggregates all the values following the semantics (see Example 11), the resulting probability is inserted in $PMap$. This will improve the performances of the computation of the probability from the BDD. Then it uses Pellet's EXPHST function [line 9] which executes the HITTINGSETTREE procedure, shown in Algorithm 4, to compute the MinAs for the query Q. These MinAs

Algorithm 7 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
```

correspond to all conflict sets found by the Hitting Set Algorithm. Pellet's EXPHST can also take as input several parameters such as the maximum number of explanations to be generated and the time limit for the inference process. If one of the limits is reached during the execution of the hitting set algorithm, Pellet 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]. JavaBDD[1] is exploited to manipulate BDDs, it is 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 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, function Prob of Algorithm 6 is called over BDD to return the probability of the query to the user.

We now prove BUNDLE correctness.

Theorem 6 (**BUNDLE correctness**). *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$(Q, \mathcal{K}, maxEx, maxTime)$ *is:*

- *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.,* Bundle$(Q, \mathcal{K}, maxEx, maxTime) \leq P(Q)$

- *equal to $P(Q)$, i.e.,* Bundle$(Q, \mathcal{K}, maxEx, maxTime) = P(Q)$ *otherwise*

[1] Available at `http://javabdd.sourceforge.net/`

Proof. Let K be ExpHST$(Q, \mathcal{K}, maxEx, maxTime)$. By Theorem 2

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

if at least a limit is set and reached and

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

otherwise. Since BUNDLE computes $P(f_K(\mathbf{X}))$ for the Boolean function

$$f_K(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(F,1) \in \kappa} X_F$$

the theorem holds. □

Example 16 (Example 14 cont.)**.** *Let us consider the KB presented in Example 9 and the query $Q = kevin : NatureLover$. We have already seen in Example 14 how BDDs are used for computing the probability of Q. In order to show how BUNDLE implements this approach first let us also consider the corresponding*

$$PMap = \{(\textit{fluffy} : Cat, 0.4), (tom : Cat, 0.3), (Cat \sqsubseteq Pet, 0.6)\}$$
$$VarAx = \emptyset$$

We recall the covering set of explanations $K = \{\kappa_1, \kappa_2\}$ where, by restricting explanations to contain only probabilistic axioms for the sake of simplicity, the explanations are

$$\kappa_1 = \{(\textit{fluffy} : Cat, 1), (Cat \sqsubseteq Pet, 1)\}$$
$$\kappa_2 = \{(tom : Cat, 1), (Cat \sqsubseteq Pet, 1)\}$$

Initially, BUNDLE initializes BDD to the `zero` *Boolean function and starts to loop over the two explanations. It enter in the inner loop considering the explanation κ_1 and initializes BDDE to the* `one` *Boolean function. Since $VarAx$ is empty, $(\textit{fluffy} : Cat, 0.4)$ is not associated with a random variable, thus the variable X_1 is created and the pair $(\textit{fluffy} : Cat, 0.4)$ is added to $VarAx$ in position 1. Function* BDDGetIthVar *is called to return in $BDDA$ a BDD corresponding to the expression $X_1 = 1$ which is then combined with $BDDE$ us-*

ing the **and** *operator. Now, the computation continues by analyzing the second axiom of* κ_1*. The axiom* $Cat \sqsubseteq Pet$ *does not have a random variable associated with it yet, so the new variable* X_2 *is created and the pair* $(Cat \sqsubseteq Pet, 0.6)$ *is added to* $VarAx$ *in position 2. The corresponding* $BDDA$ *is then generated and combined with the* $BDDE$ *which model the current explanation using the* **and** *operator. Now all the axioms in* κ_1 *have been considered, so the final* $BDDE$ *is combined with* BDD *with the* **or** *operator.*

Then BUNDLE considers κ_2 *and* $BDDE$ *is initialized to* **one**. $VarAx$ *does not contain the axiom* $Tom : Cat$ *so variable* X_3 *is created and associated to the pair* $(Tom : Cat, 0.3)$ *and* $BDDA$*, representing* $X_3 = 1$*, is joined with* $BDDE$*. The axiom* $Cat \sqsubseteq Pet$ *is found in* $VarAx$ *in position 2, so the function* BDDGetIthVar *returns in* $BDDA$ *the BDD representing* $X_2 = 1$*, which is finally combined with the current* $BDDE$*.* $BDDE$ *corresponding to the second explanation is completed so it is combined with* BDD *obtaining the one shown in Figure 14.1. Now BUNDLE calls function* Prob *which computes the probability and returns* $P(C) = 0.348$*.*

Chapter 16

TRILL

In Section 10.1.1 we showed that some tableau expansion rules are non-deterministic. When solving MIN-A-ENUM, this requires the implementation of a search strategy in an or-branching search space, because all the non-deterministic choices done by the tableau algorithm must be explored for finding all the possible explanations. In Section 10.1.1 we saw an example of such backtracking algorithm, implemented in the reasoner Pellet. However, this type of procedure must be developed in every reasoner implemented in a procedural language such as C/C++ or Java. In order to experiment with other ways to manage this non-determinism, we developed the system TRILL.

TRILL ("Tableau Reasoner for descrIption Logics in Prolog") implements a tableau algorithm in the declarative language Prolog, so the management of the rules' non-determinism is delegated to the backtracking facilities built-in in the language. TRILL is able to compute the set of all the explanations of queries w.r.t. both probabilistic and non-probabilistic KBs, and in case of a probabilistic DISPONTE KB it is able to compute the probability of queries. This is done by converting the generated explanations into a Binary Decision Diagram (BDD) which is exploited to efficiently compute the probability of the query as for BUNDLE. TRILL can answer concept membership queries and subsumption queries, and can find explanations both for the unsatisfiability of a concept contained in the KB or for the inconsistency of the entire KB.

To perform inference with TRILL, we have to convert OWL DL KBs into Prolog. To do so, we exploit a modified version of the Thea2 library [148]. Thea2 performs a direct translation of the OWL axioms into Prolog facts. For

Table 16.1: Correspondence between an OWL axiom containing a complex concept and its Prolog translation.

OWL	$forebrain\_neuron \equiv neuron \sqcap \exists partOf.forebrain$
Prolog	`equivalentClasses([forebrain_neuron,` `intersectionOf([neuron,someValuesFrom(partOf,forebrain)])])`

example, a simple subclass axiom between two named classes $Cat \sqsubseteq Pet$ is written using the `subClassOf/2` predicate as `subClassOf('Cat','Pet')`. For more complex axioms, Thea2 exploits the *list* Prolog construct, so the axiom

$$NatureLover \equiv PetOwner \sqcup GardenOwner$$

becomes

```
equivalentClasses(['NatureLover',
unionOf(['PetOwner','GardenOwner'])]).
```

Complex classes are represented by means of function symbols, as shown in Table 16.1. We modified Thea2 with respect to the management of annotations, which are used for associating probability values with axioms. When a probabilistic KB is given, for each probabilistic axiom of the form $Prob :: Axiom$, two facts are asserted, the axiom itself and an annotation assertion of the form `annotationAssertion(ProbAnnot,Axiom,literal(Prob))`, where *ProbAnnot* is the name of the annotation[1], *Axiom* is the probabilistic axiom and *Prob* is the probability value.

In order to represent the tableau, TRILL uses a pair $Tableau = (A, T)$, where

- A is a list containing information about class and role assertions with the corresponding explanation. Moreover, during initialization, for each individual `ind` in the ABox, we add the atom `nominal(ind)` to handle nominal individuals. An example of the list A is

```
[ (classAssertion(person,kevin),
    [subClassOf(man,person),classAssertion(man,kevin)]),
  (classAssertion(man,kevin),[classAssertion(man,kevin)]),
  nominal(kevin) ]
```

[1]For DISPONTE: `https://sites.google.com/a/unife.it/ml/disponte#probability`

```
safe(Ind,R,(ABox,(T,RBN,RBR))):-
  rb_lookup(R,V,RBR),
  member((X,Ind),V),
  blockable(X,(ABox,(T,RBN,RBR))),!.

safe(Ind,R,(ABox,(T,RBN,RBR))):-
  rb_lookup(R,V,RBR),
  member((X,Ind),V),
  nominal(X,(ABox,(T,RBN,RBR))),!,
  \+ blocked(Ind,(ABox,(T,RBN,RBR))).
```

Figure 16.1: Code of the predicates `safe/3`. An `R`-neighbor `Ind` of `X` is safe if (1) `X` is blockable or if (2) `X` is a nominal node and `Ind` is not blocked.

stating that *kevin* is a nominal and that it belongs to concept *man* since there is a class assertion asserting that and to concept *person* since in it belongs to *man* and *man* is a sub class of *person*.

- T is a triple (G, RBN, RBR) in which G is a directed graph that contains the main structure of the tableau, RBN is a red-black tree (a key-value dictionary) in which a key is a pair 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 in which a key is a role and its value is the set of pairs of individuals that are linked by the role.

This representation allows TRILL 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 `(A, T)`; `safe/3` takes as input also the role R. If they succeed, the corresponding state holds in the tableau. Figure 16.1 shows the code of `safe/3` while Figure 16.2 shows `indirectly_blocked/2`, where `rb_lookup/3` looks for a pair of individuals connected by the role `R`, `transpose/2` builds a transposed version `T1` of the tableau and `neighbors/3` returns the list of neighbors of `Ind` in `N`.

Tableau expansion rules are implemented following an interface; this will facilitate the insertion of new rules in the future. *Non-deterministic* rules are implemented following the interface `rule_name(Tab0, TabList)`, thus they

```
indirectly_blocked(Ind,(ABox,(T,RBN,RBR))):-
  transpose(T,T1),
  neighbors(Ind,T1,N),
  member(A,N),
  blocked(A,(ABox,(T,RBN,RBR))),!.
```

Figure 16.2: Code of the predicates `indirectly_blocked/2`. An individual `Ind` is indirectly blocked if it has at least one blocked predecessor.

take as input the current tableau `Tab0` and return the list of tableaux `TabList` created by the application of the rule to `Tab0`. *Deterministic* rules are implemented by a predicate `rule_name(Tab0,Tab)` that, given the current tableau `Tab0`, returns the tableau `Tab` obtained by the application of the rule to `Tab0`.

Figure 16.3 shows the code of the non-deterministic rule $\rightarrow \sqcup$ defined in Figure 10.1, which takes a node whose label contains a complex concept defined as a union of different concepts and creates a new tableau for each of these concepts by adding them to the label of the corresponding individual, one for each new tableau. 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.

Figure 16.4 shows a snippet of the code of the deterministic rule $\rightarrow$ *unfold*, defined in Figure 10.1, which looks for subclass and class equivalence axioms in order to add information to individuals. The predicate `unfold_rule/2` searches `(ABox0,Tabs0)`, corresponding to `Tab0`, 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. `add_nominal/4` handles nominal individuals in case `D` is a nominal concept.

Expansion rules are applied in order by `apply_all_rules/2`, first the non-deterministic ones and then the deterministic ones, as shown in Figure 16.5. The `apply_nondet_rules(RuleList,Tab0,Tab)` predicate takes as input the

```
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 16.3: Code of the $\rightarrow \sqcup$ rule. See Figure 10.1 for formal definition.

```
unfold_rule((ABox0,Tabs0),([(classAssertion(D,Ind),[Ax|Expl])|ABox],
Tabs0)):-
  find((classAssertion(C,Ind),Expl),ABox0),
  find_sub_sup_class(C,D,Ax),
  absent(classAssertion(D,Ind),[Ax|Expl],(ABox0,Tabs0)),
  add_nominal(D,Ind,ABox0,ABox).

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 16.4: Code of the $\rightarrow$ *unfold* rule. See Figure 10.1 for formal definition.

list of non-deterministic rules and the current tableau and returns a tableau obtained by the application of one of the rules. `apply_nondet_rules/3` is called as `apply_nondet_rules( [or_rule, max_rule], Tab0, Tab)`.

If a non-deterministic rule is applicable, the resulting tableau list is returned by the predicate corresponding to the applied rule, a cut avoids backtracking to other possible choices for the non-deterministic rules and the `member/2` predicate is used to non-deterministically choose a tableau from the list.

If no non-deterministic rule is applicable, deterministic rules are tried sequentially by the predicate `apply_det_rules/3`, shown in Figure 16.5, that is called as `apply_det_rules(RuleList,Tab0,Tab)`. This predicate takes as input the list of deterministic rules in `RuleList` and the current tableau and returns a tableau obtained by the application of one of the rules. After the application of a deterministic rule, a cut is performed to avoid backtracking to other rule choices. If no rule is applicable, the input tableau is returned and rule application stops, otherwise a new round of rule application is executed.

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 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 its expansion. 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.

Once the set of explanations is found, TRILL executes the `compute_prob/2` predicate, shown in Figure 16.6, which takes the set of explanations and builds the BDDs by using the `build_bdd/3` predicate, shown in Figure 16.6. The `build_bdd/3` predicate scans each explanation and, for each variable in the current explanation, searches for the corresponding probabilistic axiom using the predicate `get_prob_ax/3`. Finally, it computes the probability of the query from the BDD so built using `ret_prob/3`. `one/2` and `zero/2` return BDDs representing the Boolean constants 1 and 0; `and/4` and `or/4` execute Boolean operations between BDDs. `get_var_n/5` returns the random variable `V` associated with axiom `AxN` and the list of probabilities `[Prob,ProbN]`,

```
apply_all_rules(Tab0,Tab):-
  apply_nondet_rules([or_rule,max_rule],Tab0,Tab1),
  (Tab0=Tab1 ->
    Tab=Tab1
  ;
    apply_all_rules(Tab1,Tab)
  ).

apply_nondet_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,Tab).

apply_nondet_rules([H|T],Tab0,Tab):-
  call(H,Tab0,L),!,
  member(Tab,L),
  dif(Tab0,Tab).

apply_nondet_rules([_|T],Tab0,Tab):-
  apply_nondet_rules(T,Tab0,Tab).

apply_det_rules([],Tab,Tab).

apply_det_rules([H|_],Tab0,Tab):-
  call(H,Tab0,Tab),!.

apply_det_rules([_|T],Tab0,Tab):-
  apply_det_rules(T,Tab0,Tab).
```

Figure 16.5: Application of the expansion rules: predicates `apply_all_rules/2`, `apply_nondet_rules/3` and `apply_det_rules/3`.

```
compute_prob(Expl,Prob):-
  retractall(v(_,_,_)),
  retractall(na(_,_)),
  retractall(rule_n(_)),
  assert(rule_n(0)),
  init_test(_,Env),
  build_bdd(Env,Expl,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 16.6: Code of the predicates `compute_prob/2` and `build_bdd/3`.

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 imported from a foreign Prolog library of the `cplint` suite [118].

TRILL is available for Yap Prolog[2] [127] and SWI-Prolog[3] [154]. In particular, SWI-Prolog is the basis of the TRILL on SWISH web application described in Section 16.1. Using SWI-Prolog, TRILL can be installed by the user with the goal `pack_install(trill)`. After this call, TRILL can be loaded with the command `use_module(library(trill))`. The code of TRILL is available at `https://sites.google.com/a/unife.it/ml/trill`.

16.1 TRILL on SWISH

In order to popularize TRILL, we implemented a Web application which embeds the reasoner and allows users to try it and collaborate using a Web browser, without the need to install anything on the client machine. The application is called "TRILL on SWISH" and is available at `http://trill.lamping.unife.it`.

TRILL on SWISH is based on SWISH[4] [80], a web application using various features and packages of SWI-Prolog that allows the users to write Prolog programs and ask queries through the browser. The SWISH page allows the insertion of Prolog programs and queries via a text editor, then, it collects the text in the program editor and the query and sends this information to the server, which creates a Pengine (Prolog Engine). The Pengine initializes a temporary private module in which the program is compiled, then it checks whether the query execution is safe. If executing the query may compromise the system, an error is returned, otherwise the query is computed in a "sandboxed" environment and the results are returned to the user through JSON messages. Sandboxing ensures that only predicates that do not have side effects, such as accessing the file system or loading foreign extensions, are called.

SWISH uses the SWI-Prolog Pengines library [80], which allows creating Prolog engines from either an ordinary Prolog thread, another Pengine, or

[2] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[3] `http://www.swi-prolog.org/`
[4] `http://swish.swi-prolog.org/`

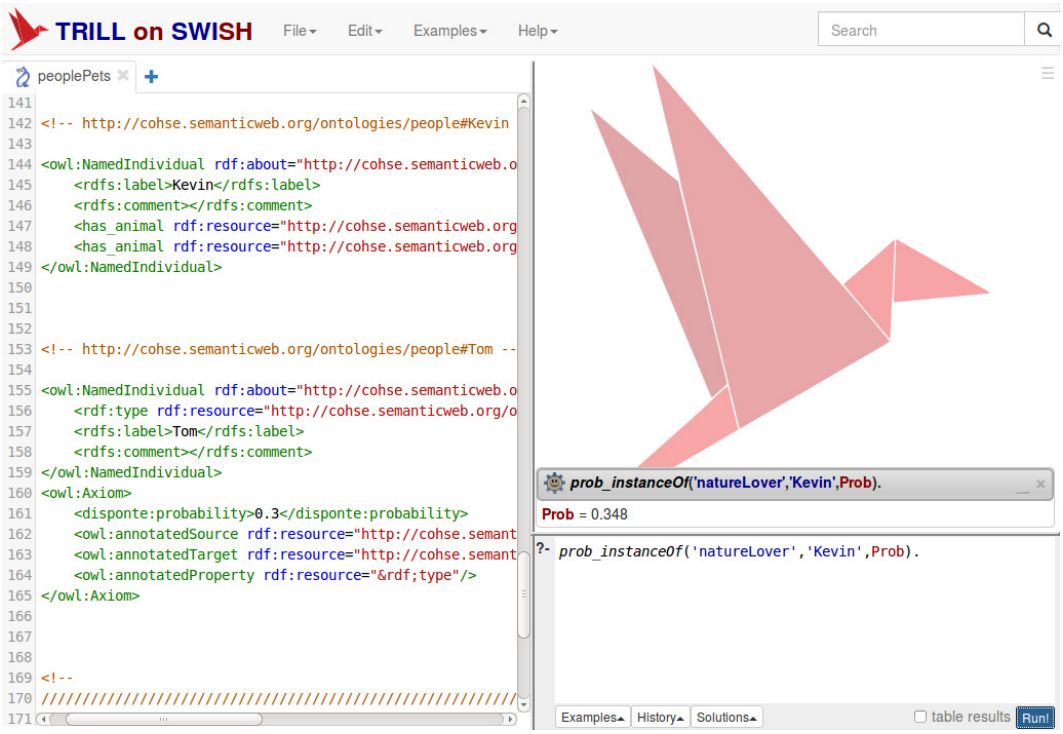


Figure 16.7: "TRILL on SWISH" web interface.

JavaScript running in a web client. Each Pengine is associated with a Prolog thread with two private message queues, one for incoming requests and one for outgoing responses, and a private dynamic clause database.

The SWISH web server is implemented by the SWI-Prolog HTTP package, a series of libraries for serving data on HTTP [153].

For TRILL on SWISH we used the version of SWISH included in ClioPatria[5], a Semantic Web server based on SWI-Prolog which offers features that allow handling RDF.

TRILL on SWISH, whose interface is shown in Figure 16.7, allows the user to write a KB in the RDF/XML format in the left panel and write a query in the bottom right panel. Both the KB and query editor have syntax highlighting. Moreover, URIs in queries can be written without the base URI or using a namespace defined in the RDF/XML file: the system checks for possible misspellings of URIs that are reported to the user.

In case one needs KB serializations different from RDF/XML or prefers a

[5] `http://cliopatria.swi-prolog.org/home`

GUI to build the KB, it is possible to use WebProtégé [143] to develop the KB, then download it in RDF/XML and upload it into TRILL on SWISH. Currently, we are working on the integration of the two systems in order to allow users to modify the KB in both systems moving from one to the other by just pressing a button.

We tested the robustness of the application by running two different stress tests. First, we submitted queries without imposing a time limit for the executions. The queries and the KBs were chosen in order to saturate the main memory. Then, we set the time limit to 300 seconds, and we ran again all the queries. In both cases, the serves simply kills or interrupts the thread that exhausts the memory or that reaches the time limit, without affecting the executions of other threads. An error message is returned to the client regarding the motivation for the execution interruption. What is important to bear in mind is that TRILL on SWISH is a testing tool useful for developing and experimenting also in a collaborative way, but it is not befitting heavy computations, for which a local installation should be used. For these reasons, and to ensure the server responsiveness, we imposed a time limit of 300 seconds on query execution. These tests show that the system is robust and can manage high loads even in case of errors in some threads.

Chapter 17

TRILL^P

TRILL^P (for "Tableau Reasoner for descrIption Logics in Prolog powered by Pinpointing formula") is based on the reasoner TRILL but, differently from it, TRILL^P solves the MIN-A-ENUM problem following the approach shown in Section 10.1.2. Therefore, TRILL^P builds a pinpointing formula representing the set of explanations for the query. Then the pinpointing formula can be directly translated into a BDD.

Since TRILL^P is based on TRILL, it inherits all its features. Thus, the representation of the tableau and the management of the blocking system is the same of TRILL with some minor differences due to the different representation of the explanations. For example, in the representation of the tableau by a pair (A, T), the list A contains for each class assertion the pinpointing formula instead of the list of explanations. In TRILL^P the pinpointing formula is encoded by means of the combination of the predicates `and/1` and `or/1`. For example, the Boolean formula $((F_2 \wedge F_4) \vee (F_3 \wedge F_5)) \wedge F_6 \wedge F_1$ from Example 2 is modeled as `and([or([and([F2,F4]),and([F3,F5])]),F6,F1])`.

As seen in Section 10.1.2, the algorithm for computing the pinpointing formula is limited to $\mathcal{ALC}$ DL KBs, thus TRILL^P uses a subset of the rules implemented in TRILL. `apply_nondet_rules/3` and `apply_det_rules/3` predicates are called as shown in Figure 17.1 which contains a snippet of the code for executing the tableau expansion rules. For the sake of simplicity the figure contains only the definition of the predicates that have been modified in TRILL^P.

In Figure 10.1, the symbol $(*)$ denotes the rules used by TRILL^P. In these

```
apply_all_rules(Tab0,Tab):-
  apply_nondet_rules([or_rule],Tab0,Tab1),
  (Tab0=Tab1 ->
    Tab=Tab1
  ;
    apply_all_rules(Tab1,Tab)
  ).

apply_nondet_rules([],Tab0,Tab):-
  apply_det_rules([and_rule,unfold_rule,
  add_exists_rule,forall_rule,
  exists_rule],Tab0,Tab).
```

Figure 17.1: A snippet of the code for the application of the expansion rules by means of `apply_all_rules/2`, `apply_nondet_rules/3` and `apply_det_rules/3`. What one should note is the difference in the rule lists with those of TRILL, shown in Figure 16.5

rules, τ associates the pinpointing formula to label of the class assertion, while the operator $\cup$ for τ joins two Boolean formulas with the `OR` Boolean operator. Moreover, when a concept is already present in a node label, TRILL^P checks whether to update the tracing function by performing a $\psi-$*insertability* test. This test is done by means of a satisfiability solver. In particular, TRILL^P conjoins the negation of the pinpointing formula contained in the label of the individual in the tableau with the Boolean formula we want to add to the label and tests the satisfiability of such formula. This step is performed by the `test/2` predicate shown in Figure 17.2. The predicate `test/2` first calls `build_f/3` which takes two Boolean formulas `L1` and `L2` and creates the conjunction that will be tested by means of the satisfiability solver. Predicates `cnf/2` and `sat/1` are defined in Prolog built-in libraries providing the interface to a SAT solver. These libraries were originally presented in [27], where the authors described the implementation of an interface between Prolog and the MiniSat SAT solver [39], a small (about 1200 lines of C code) and efficient SAT solver. Predicate `cnf/2` converts a propositional formula `F`, in which the Boolean operators `and`, `or` and `not` are represented by `*`, `+` and `-` respectively, into a conjunctive normal form `Cnf`. Finally, `sat/1` takes as input such a conjunctive normal form formula and succeeds if it is satisfiable. If the test

```
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 17.2: Definition of the predicates `test/2` and `build_f/3`.

returns true, TRILL^P combines the two Boolean formulas with the `OR` Boolean operator.

TRILL^P, differently from TRILL, computes directly a pinpointing formula which is a monotone Boolean formula that represents the set of all MinAs. Once the pinpointing formula is built, we can apply knowledge compilation and *directly transform it into a Binary Decision Diagram (BDD)*, from which we can compute the probability of the query in a way similar to that of TRILL shown in Figure 16.6. In TRILL^P the predicates `bdd_or/3` and `bdd_and/3` in order to convert general pinpointing formulas to BDDs.

TRILL^P is available for Yap Prolog[1] [127]. The code can be obtained from `https://sites.google.com/a/unife.it/ml/trill`.

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

Chapter 18

Complexity of Inference

We start the discussion about the complexity of the algorithms presented in this Part from the results of Jung and Lutz [67] on the problem of computing the probability of conjunctive queries to probabilistic databases in the presence of an ontology. In their settings, the TBox is certain while the ABox can contain probabilistic axioms which are then associated with pairwise independent Boolean random variables. Even in the presence of an $\mathcal{ELI}$ non-probabilistic TBox (less expressive than DL-Lite), only very simple conjunctive queries can be answered in PTime, while most queries are #P-hard.

The class #P [145] describes counting problems associated with decision problems in NP. More formally, #P is the class of function problems of the form "compute $f(x)$", where f is the number of accepting paths of a nondeterministic Turing machine running in polynomial time. A prototypical #P problem concerns the computation of the number of satisfying assignments of a conjunctive normal form (CNF) Boolean formula. #P problems were shown very hard. First, a #P problem must be at least as hard as the corresponding NP problem. Second, [141] showed that a polynomial-time machine with a #P oracle ($P^{\#P}$) can solve all problems in PH, the entire polynomial hierarchy.

In DISPONTE, if we restrict to probabilistic axioms in the ABox only we are same settings considered in [67], thus these complexity results, provide a lower bound for DISPONTE.

In order to investigate the complexity of the three systems presented in the previous chapters, we can consider the two problems that they solve for answering a query separately. The first one is axiom pinpointing in both its ver-

sions, i.e., finding the set of all MinAs or finding the pinpointing formula. The computational complexity of the first version has been studied in a number of works [102, 103, 104]. Baader et al. [6] showed that there can be exponentially many MinAs for a very simple DL that allows only concept intersection.

Example 17. *Given an integer $n \geq 1$, consider the TBox*

$$\mathcal{T}_n = \{B_{i-1} \sqsubseteq P_i \sqcap Q_i, P_i \sqsubseteq B_i, Q_i \sqsubseteq B_i | 1 \leq i \leq n\}$$

The size of $\mathcal{T}_n$ is linear in n and $\mathcal{T}_n \models B_0 \sqsubseteq B_n$. There are 2^n MinAs for $B_0 \sqsubseteq B_n$ since, for each $i, 1 \leq i \leq n$, it is enough to have $P_i \sqsubseteq B_i$ or $Q_i \sqsubseteq B_i$ in the set.

The number of explanations for $\mathcal{SROIQ}(\mathbf{D})$ may be even larger. Given this fact, we do not consider complexity with respect to the input only. Corollary 15 in [104] shows that MIN-A-ENUM cannot be solved in *output polynomial time* for *DL-Lite*$_{bool}$ TBoxes unless $P = NP$. An algorithm runs in output polynomial time [66] if it computes all the output in time polynomial in the overall size of the input and the output. If we consider $\mathcal{SROIQ}(\mathbf{D})$, the results obtained for *DL-Lite*$_{bool}$ also hold, since it is a sublogic of $\mathcal{SROIQ}(\mathbf{D})$. When explicitly considering the problem of finding a pinpointing formula, Corollary 3 in [4] 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 to be solved is computing the probability of a query. This problem can be reduced to computing the probability of a SUM-OF-PRODUCTS.

Definition 3 (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.*

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 (see e.g. [111]), hence computing the probability of an axiom from a $\mathcal{SHOIN}(\mathbf{D})$ knowledge base is intractable.

However, the algorithms proposed for solving the two problems were shown to be able to work on inputs of real world size. For example, all MinAs have been found for various entailments over many real world ontologies within a few seconds [68, 70]. As regards 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 [34, 117, 118, 123, 73, 125, 124, 119, 120]). Also methods for weighted model counting [128, 23] can be used to solve the SUM-OF-PRODUCTS problem.

Moreover, Section 20 shows that in practice our algorithms can compute the probability of entailments on KBs of real-world size.

Chapter 19

Related Inference Systems

Despite the large number of proposals for combining probability and DLs and the even larger availability of DL reasoner systems, there is a lack of systems that perform probabilistic inference on probabilistic DLs. One of the first probabilistic reasoners is PRONTO [74]. Similarly to BUNDLE, this system is based on Pellet, but differently from it, PRONTO exploits also a linear program solver such as GLPK[1] in order to execute inference on P-$\mathcal{SHIQ}(\mathbf{D})$ [88] KBs. In these KBs, as already seen in Chapter 13, 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.

ELOG [99] is a reasoner for the log-linear DL $\mathcal{EL}^{++}$-LL. As seen in Chapter 13, this semantics combines DLs with probabilistic log-linear models to associate a real-valued weight, which defines a degree of confidence, to any axiom of a $\mathcal{EL}^{++}$-LL KB. ELOG casts inference as an optimization problem, similarly to PRONTO it first transforms the inference problem into an integer linear program. Then it applies cutting plane inference in order to restrict the optimization problem: first it solves the optimization problem without some constraints and then it iteratively adds the constraints that are violated by the previously found solution until no more constraints can be added.

[1] `https://www.gnu.org/software/glpk/`

BUNDLE differs from these systems in the probabilistic semantics considered and in the fact that it does not exploit linear program solvers.

Looking beyond the combination of probability theory with DLs, *fuzzyDL* [15] is a DL reasoner supporting fuzzy logic reasoning implemented in Java. It combines a tableaux algorithm and a Mixed Integer Linear Programming (MILP) optimization problem to execute inference on fuzzy DLs [139, 140], which are extensions to classical DLs for allowing the specification of *fuzzy* (or *vague*, *imprecise*) concepts. For a survey on fuzzy DLs we refer the reader to [89]. This system differs from BUNDLE not only in the use of a linear program solver, but also because it performs inference on fuzzy KBs whose underlying uncertainty basis is completely different from those of DISPONTE.

Differently from BUNDLE and from the systems presented above, reasoners written in Prolog can exploit Prolog's backtracking facilities for performing the search in presence of non-deterministic operations. This has been observed in various works. Beckert and Posegga [10] proposed a tableau reasoner in Prolog for First Order Logic (FOL). It is based on free-variable semantic tableaux but it is not tailored to DLs.

In [63] the authors presented the KAON2 algorithm. It exploits a refutational theorem proving method for FOL with equality, called basic superposition, and a new inference rule, called decomposition, in order to reduce a $\mathcal{SHIQ}$ KB into a disjunctive datalog program.

DLog [86] is able to execute ABox reasoning for the $\mathcal{SHIQ}$ language. It allows storing the content of the ABox externally in a database and answering instance check and instance retrieval queries by transforming the KB into a Prolog program.

Meissner [94] presented the implementation of a Prolog reasoner for the DL $\mathcal{ALCN}$. This work was extended by the work of Herchenröder [59], which considered $\mathcal{ALC}$ and improved the work of Meissner by implementing heuristic search techniques to reduce the running time. Faizi [40] added to [59] the possibility of returning information about the steps executed during the inference process but still handled only $\mathcal{ALC}$.

A different approach is the one of Ricca et al. [115] that presented OntoDLV, a system for reasoning on a logic-based ontology representation language called OntoDLP. This is an extension of (disjunctive) Answer Set Pro-

gramming (ASP) and can interoperate with OWL. OntoDLV, after rewriting the OWL KB into the OntoDLP language, can retrieve information directly from external OWL Ontologies and uses ASP to answer queries.

In [49] and [48], 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.

All these systems do not return the set of explanations, thus they cannot be applied to our problem. Moreover, TRILL differs from the previous works for the target description logics ($\mathcal{ALC}$). Finally, both TRILL and TRILL^P differ from DLog for the possibility of answering general queries instead of instance check and instance retrieval only. Note that all the above mentioned logic programming systems are not probabilistic reasoners, hence they are not able to compute probability of queries and for this reasons they require extensions to deal with uncertainty.

FOProbLog [18] is an extension of ProbLog. Similarly to our systems, it follows the distribution semantics and exploits BDDs to compute the probability of queries. Nonetheless, FOProbLog is a reasoner for FOL not tailored to DLs, so the algorithm could be suboptimal for them. Moreover it cannot manage probabilistic facts which are annotated with more than one probability value.

Chapter 20

Experiments

In order to test the performance of our reasoning systems we performed several experiments. All the experiments have been performed on Linux machines with a 3.10 GHz Intel Xeon E5-2687W. For BUNDLE we allotted 2GB memory to Java. In the following, each section presents a different test. At the end, Section 20.6 discusses the results.

20.1 BUNDLE: Comparison with PRONTO

We compared BUNDLE with PRONTO by running queries w.r.t. increasingly complex ontologies.

In the first experiment, following the approach shown in [75], we generated the test ontologies by randomly sampling axioms from a large probabilistic ontology that models breast cancer risk assessment (BRCA). The main idea behind the design of the ontology was to reduce risk assessment to probabilistic entailment in P-$\mathcal{SHIQ}(\mathbf{D})$. The BRCA ontology contains a certain part and a probabilistic part. The tests were defined by randomly sampling axioms from the probabilistic part of this ontology which were then added to the certain part in order to create samples which are probabilistic KBs containing the full certain part of the BRCA ontology and a subset of the probabilistic constraints. The number of these constraints varied from 9 to 15, and, for each number, we generated 100 different consistent ontologies.

In order to generate a query, we added an individual a to the ontology that is randomly assigned to each class that appears in the sampled conditional

constraints with probability 0.6. If the class is composite, as for example *PostmenopausalWomanTakingTestosterone*, a is assigned to the component classes rather than to the composite one. In the example above, a will be added to *PostmenopausalWoman* and *WomanTakingTestosterone*.

The ontologies were translated into DISPONTE by replacing each constraint $(D|C)[l, u]$ with the axiom $u :: C \sqsubseteq D$. For instance, the statement that an average woman has up to 12.3% chance of developing breast cancer in her lifetime expressed by

$$(WomanUnderAbsoluteBRCRisk|Woman)[0, 0.123]$$

is translated into

$$0.123 :: WomanUnderAbsoluteBRCRisk \sqsubseteq Woman$$

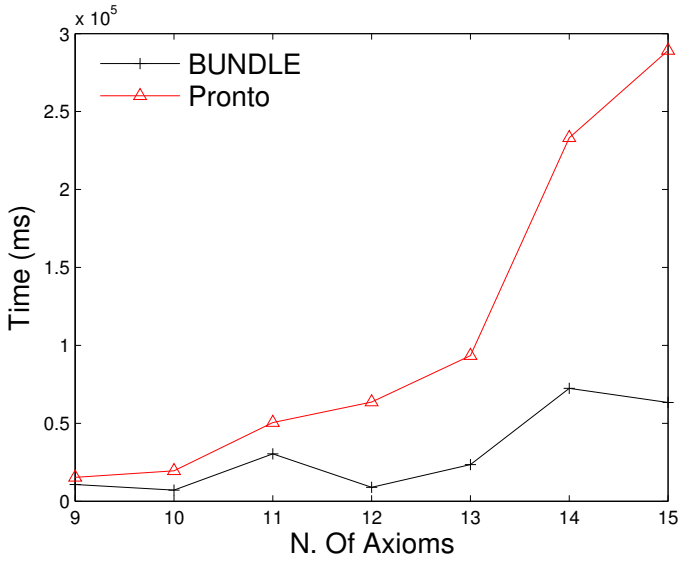
For each ontology, we randomly selected a class C among those that represent women under increased and lifetime risk such as *WomanUnderLifetimeBRCRisk* and *WomanUnderStronglyIncreasedBRCRisk* and performed the query $a : C$. We then applied both BUNDLE and PRONTO to each generated test and we measured the execution time for inference and the memory used. Figure 20.1a shows the execution time averaged over the 100 KBs as a function of the number of probabilistic axioms and, similarly, Figure 20.1b shows the average amount of memory used. As one can see, inference times are similar for small knowledge bases, while the difference between the two reasoners rapidly increases for larger knowledge bases. The memory usage for BUNDLE is always less than 53% that of PRONTO.

A second test was performed over larger KBs, following the method of [76]. We considered three different datasets:

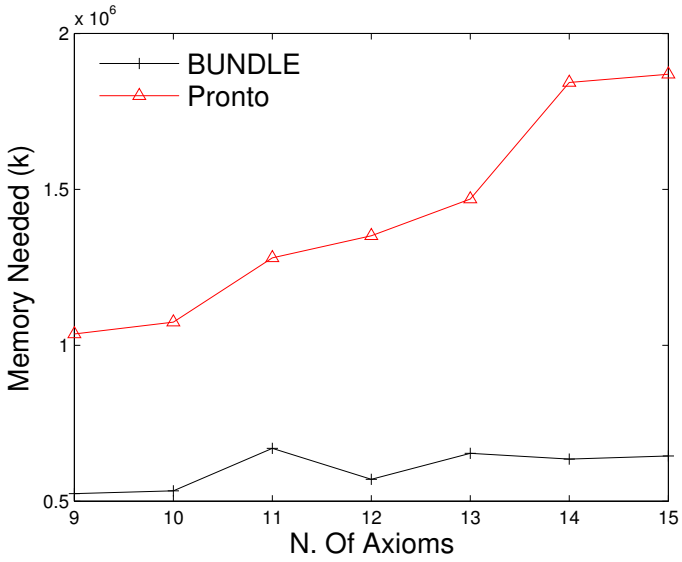
- an extract from the Cell[1] ontology which represents cell types of the prokaryotic, fungal, and eukaryotic organisms;

- an extract from the NCI Thesaurus[2] that describes human anatomy;

[1] `http://cellontology.org/`
[2] `http://ncit.nci.nih.gov/`



(a) Average execution time (s) for inference.



(b) Average memory consumption (Kb) for inference.

Figure 20.1: Comparison of average execution time and memory consumption between BUNDLE and PRONTO on the BRCA KB.

Table 20.1: BUNDLE's average execution time and number of executions terminated with a time-out (TO) for the queries to the Cell, Teleost and NCI KBs. The first column reports the expressiveness of each KB and the size of the non-probabilistic TBox.

		Size of the Probabilistic TBox				
Dataset & Infos		0	250	500	750	1,000
Cell	runtime (s)	0.76	2.84	3.88	3.94	4.53
$\mathcal{ALE}+$, 1,263 TBox axioms	TO	0	28	39	50	55
Teleost	runtime (s)	2.11	8.87	31.80	33.82	36.33
$\mathcal{ALEI}+$, 3,406 TBox axioms	TO	0	7	32	32	44
NCI	runtime (s)	3.02	11.37	11.37	16.37	24.90
$\mathcal{ALE}+$, 5,423 TBox axioms	TO	0	1	24	23	36

- an extract from the Teleost_anatomy[3] ontology (Teleost for short) that is a multi-species anatomy ontology for teleost fishes.

For each of these KBs, we considered the versions of increasing size used by [76]: they add 250, 500, 750 and 1,000 new probabilistic conditional constraints to the publicly available non-probabilistic version of each ontology. We converted these KBs into DISPONTE in the same way as for the BRCA ontology and we created a set of 100 different random subclass queries for each KB. The generation of the queries was made by building the hierarchy of each KB and randomly selecting two classes connected in the hierarchy, so that each query had at least one explanation. We imposed a time limit of 5 minutes for BUNDLE to answer each query. If this limit is reached, BUNDLE's answer is "time-out".

In Table 20.1 we report, for each version of the datasets, the average execution time and the number of queries that terminated with a time-out (TO) for BUNDLE. The averages are computed on the queries that did not end with a time-out. In addition, we report the expressiveness and the number of non-probabilistic TBox axioms of each KB. In all these cases, PRONTO terminates with an out-of-memory error.

As can be seen, BUNDLE can manage larger KBs than PRONTO due to the lower amount of memory needed, as confirmed by the previous tests on

[3] `http://phenoscape.org/wiki/Teleost_Anatomy_Ontology`

Table 20.2: BUNDLE's average execution time for the queries without explanations to the Cell, Teleost and NCI KBs.

		Size of the Probabilistic TBox				
Dataset		0	250	500	750	1,000
Cell	runtime (s)	0.47	1.52	2.43	2.52	3.14
Teleost	runtime (s)	1.15	4.51	12.76	14.69	15.27
NCI	runtime (s)	1.42	4.15	5.99	7.41	7.63

BRCA. Moreover, BUNDLE answers most queries in a few seconds. However, some queries have a very high complexity that causes BUNDLE to reach the time-out, confirming the complexity results. In these cases, since the time-out is reached during the computation of the explanations, limiting the number of explanations is necessary, obtaining a lower bound on the probability that becomes tighter as more explanations are allowed.

20.2 BUNDLE: Not Entailed Queries

Regarding BUNDLE, in a further test we investigated cases for which subsumption does not hold. Thus, for the same versions of increasing size of the Cell, NCI and Teleost KBs we randomly created 100 different subclass queries that do not have explanations. In Table 20.2 we report, for each KB, the runtime in seconds. As for the previous test, we set a time-out of 5 minutes but this limit was never reached.

20.3 BUNDLE: Inference with Limited Number of Explanations

We studied how the execution time and the probability of queries vary when imposing a limit on the number of explanations.

We chose the Grid[4] KB that is part of the myGrid project. The Grid KB has already been used for testing the performances of Pellet in [70]. It

[4] `http://www.myGrid.org.uk/`

belongs to the bioinformatics domain and contains concepts at a high level of abstraction. For the test, we used a version of the Grid KB with $\mathcal{SHOIN}$ expressiveness that contains 2,838 axioms, 550 atomic concepts, 69 properties and 13 individuals, downloaded from the Tones repository[5]. We associated a probability of 0.5 to each axiom of the KB and then we ran 100 different subclass queries. The queries were created as in the previous sections, first the hierarchy of the KB was computed and then concepts were randomly selected in order to create queries with at least one explanation.

We first computed the correct probability of each query by using BUNDLE without a limit on the number of explanations. Then we ran each query several times, each time with an increasing limit. The maximum number of explanations is 16: there were 20 queries with 16 explanations but most of the queries have a number of explanations between 1 and 5. The value of the limit was varied from 2 to 16 with step 2. We computed the relative error e between the correct probability p of a query and the probability p' returned by BUNDLE with a limit on the number of explanations with the formula $e = \frac{p-p'}{p}$. Then we averaged the relative error over all the queries.

In Figure 20.2 we show how the mean relative error varies with respect to the limit on the number of explanations. As can be seen, the quality of the answer increases as the limit on the number of explanations increases. Table 20.3 reports the execution times, averaged over all the 100 queries. The row of Table 20.3 with "–" in the first column contains the average execution time for BUNDLE without a limit on the number of explanations. Figure 20.3 shows the execution time as a function of the limit on the number of explanations, based on the values of Table 20.3.

20.4 BUNDLE: Scalability

For testing the scalability of BUNDLE, we considered two different KBs: the full version of the NCI Thesaurus (NCI_full for short) with $\mathcal{SH}$ expressiveness that contains 3,382,017 axioms, and the Foundational Model of Anatomy Ontology (FMA for short)[6] with $\mathcal{ALCOIN}(\mathbf{D})$ expressiveness. FMA is a KB

[5] `http://rpc295.cs.man.ac.uk:8080/repository/browser`
[6] `http://sig.biostr.washington.edu/projects/fm/index.html`

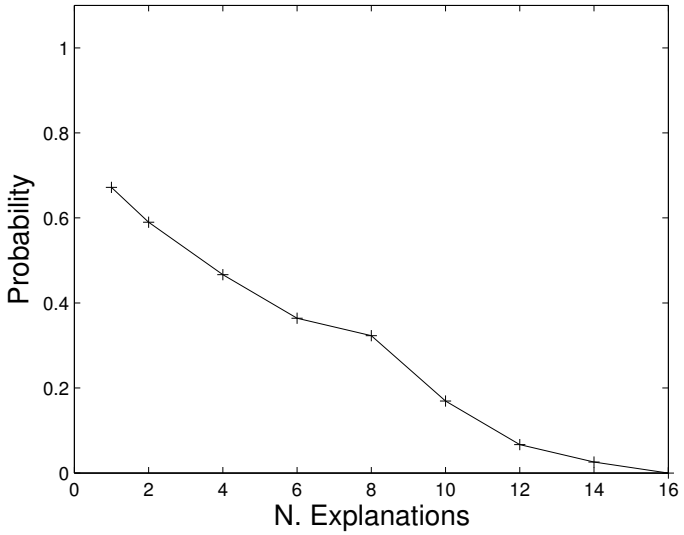


Figure 20.2: Mean relative error of the probability of queries computed with BUNDLE as a function of the limit on the number of explanations for the Grid KB.

Table 20.3: BUNDLE's average execution time depending on the limit on the number of explanations for the Grid KB. The last row reports the execution time spent for finding the set of all explanations when no limits are imposed.

Limit on the explanations	Runtime (s)
0	0.81
2	1.40
4	1.44
6	1.46
8	1.49
10	1.52
12	1.55
14	2.10
16	9.36
–	18.44

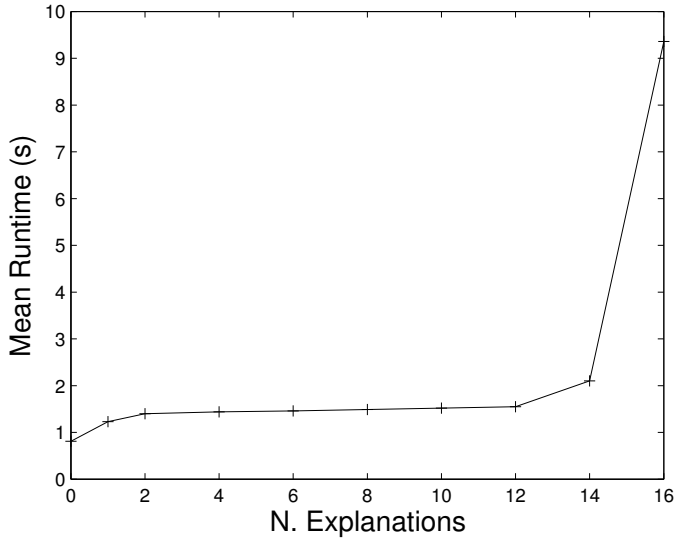


Figure 20.3: BUNDLE's average execution time (s) as the limit on the number of explanations to the queries varies for the Grid KB.

for biomedical informatics that models the phenotypic structure of the human body anatomy. It contains 88,252 axioms in the TBox and RBox and 237,382 individuals.

For NCI_full we generated 10 ontologies of increasing size that contain $10\%, ..., 100\%$ of the axioms. Then we randomly selected an increasing number of certain axioms from these subontologies and made them probabilistic. We sampled 5,000, 10,000, 15,000, 20,000, 25,000 different probabilistic axioms, obtaining 50 different probabilistic KBs with total size from 338,201 to 3,382,017 axioms. Then we randomly created 100 subclass queries for each of the 50 subontologies and ran them. Figure 20.4 shows the trend of the runtime averaged over the queries with respect to the total size of the ontologies and the subset of probabilistic axioms. The maximum time spent for computing the probability of a query is 266.24 seconds with the KB that contains 90% of the axioms.

Finally, we exploited the FMA ontology for running a scalability test where only the size of the ABox varies. We generated versions of the ontology that contain the entire TBox and RBox, 500 probabilistic axioms and an increasing number of individuals. The size of the ABox varies between 50,000 and all the axioms contained in the full ABox with a step of 50,000. We generated

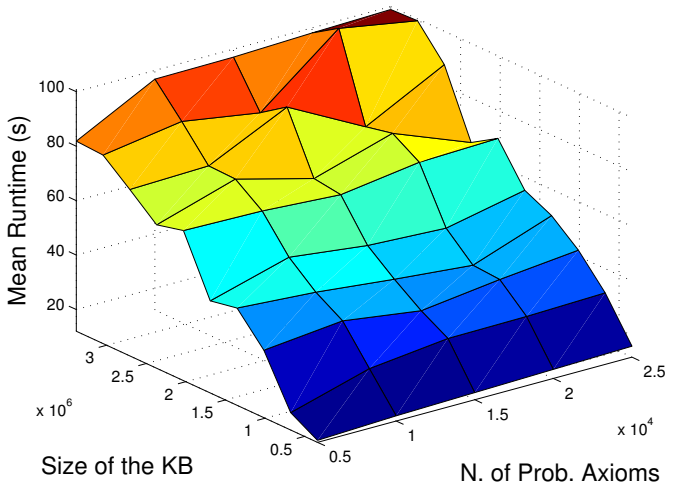


Figure 20.4: BUNDLE's average execution time (s) for the queries to the NCI_full KB on versions of increasing size of the ontology and of the probabilistic part. The x axis contains the total number of axioms of the KB (including the probabilistic ones) while the y axis contains the number of probabilistic axioms.

100 instance-of queries by randomly selecting an individual and a class among those to which it belongs. Figure 20.5 shows how the runtime averaged over the queries varies with respect to the size of the ABox. With 237,382 individuals, which correspond to the entire ABox, BUNDLE raises an out-of-memory error. The maximum inference time reached is 298.98 seconds with 200,000 individuals.

20.5 TRILL, TRILL^P & BUNDLE: Comparing Different Approaches

In order to evaluate the performances of TRILL and TRILL^P, we compared them with BUNDLE. We used four different knowledge bases of various complexity:

- BRCA[7] used in the comparison with PRONTO;

[7] `http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl`

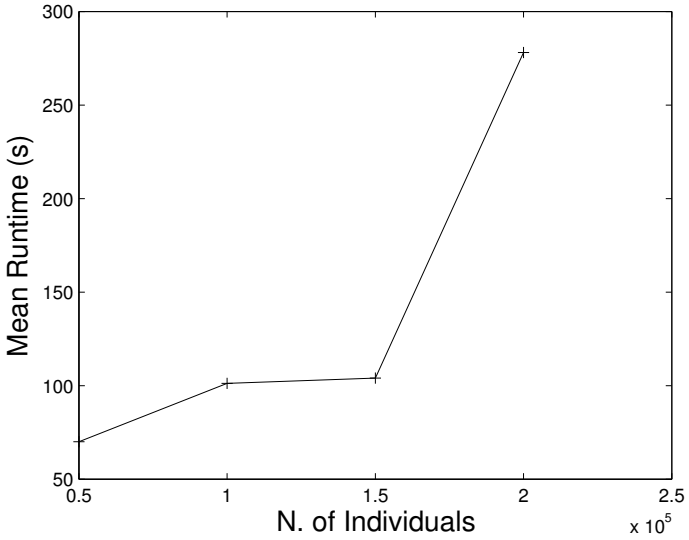


Figure 20.5: BUNDLE's average execution time (s) for the queries to the FMA KB with respect to the increasing size of the ABox. BUNDLE cannot manage the entire ABox (237,382 individuals).

- an extract of the DBPedia[8] ontology obtained from Wikipedia;
- Biopax level 3[9], which models metabolic pathways;
- Vicodi[10], which contains information on European history.

For the tests, we used a version of the DBPedia and Biopax KBs without the ABox and a version of BRCA and of Vicodi with an ABox containing 1 individual and 19 individuals respectively. We added 50 probabilistic axioms to each KB. The probability values were learned using EDGE (see Section 22.2), a system that computes the probability associated with axioms starting from a set of positive and negative examples.

For each dataset we randomly created 100 different queries. In particular, for the DBPedia and Biopax datasets, we created 100 subclass-of queries, while for the other KBs we created 80 subclass-of and 20 instance-of queries. For generating the subclass-of queries, we randomly selected two classes that are connected in the hierarchy of classes, so that each query had at least one

[8] `http://dbpedia.org/`
[9] `http://www.biopax.org/`
[10] `http://www.vicodi.org/`

Table 20.4: Expressiveness, average number of MinAs and average time (in seconds) for computing the probability of queries with the reasoners TRILL, TRILL^P and BUNDLE.

Dataset	Expres.ness	avg. n. of MinAs	TRILL time (s)	TRILL^P time (s)	BUNDLE time (s)
BRCA	$\mathcal{ELH}$	6.49	27.87	4.74	6.96
DBPedia	$\mathcal{EL}$	16.32	51.56	4.67	3.79
Biopax level 3	$\mathcal{SHOIN}(\mathbf{D})$	3.92	0.12	0.12	1.85
Vicodi	$\mathcal{ALH}(\mathbf{D})$	1.02	0.19	0.19	1.12

explanation. For the instance-of queries, we randomly selected an individual a and a class to which a belongs by following the hierarchy of the classes, starting from the classes to which a explicitly belongs in the KB.

Table 20.4 shows, for each ontology, the average number of different MinAs computed and the average time in seconds that TRILL, TRILL^P and BUNDLE take for computing the probability of the queries. In particular, BRCA and the version of DBPedia used contain a large number of subclass axioms between complex concepts.

20.6 Discussion

BUNDLE has been compared with the probabilistic reasoner PRONTO and tested for scalability on several real world KBs. The experiments show that BUNDLE is able to deal with ontologies of significant complexity due to the low amount of memory needed. Moreover, BUNDLE answers most queries in a few seconds. In case of high complexity queries, a limit on the running time or on the number of explanations to find can be set, in this case BUNDLE returns an approximate value of the probability of the query. The value is a lower bound that becomes tighter as more explanations are found.

Encouraged by the good results obtained by BUNDLE, we compared its performances with those of TRILL and TRILL^P in order to evaluate them. The results of the comparison show that the performances of TRILL and TRILL^P are comparable with and sometimes better than those of BUNDLE, even if they lack all the optimizations that BUNDLE inherits from Pellet. In

particular, when a KB is relatively simple, BUNDLE's higher cost is due to its expensive initialization phase that is not present in TRILL and TRILL^P, while this initialization phase becomes more effective in case of more complex KBs. These results represent evidence that a Prolog implementation of Semantic Web tableau reasoners is feasible and that may lead to practical systems. Moreover, TRILL^P provides an improvement of the execution time with respect to TRILL when more MinAs are present.

Part V

Learning in Probabilistic DLs

Chapter 21

Learning

In Chapter 11 we have discussed the problem of learning in probabilistic logic programming (PLP). We have seen that the learning problem can be divided in two different sub-problems: (1) *parameter learning* and (2) *structure learning*.

Parameter learning takes as input a KB and a set of positive and negative examples and returns the parameters for the axioms in the KB such that they maximize a scoring function, while structure learning, given the same inputs, returns a new KB containing new axioms together with their parameters which maximizes the scoring function.

Probabilistic logics are of foremost importance also in the Semantic Web since uncertain information is ubiquitous in real world domains and in the resources available on the Web, due to methods used for collecting data and to the inherently distributed nature of the data sources. It is thus very important to develop probabilistic DLs so that the uncertainty is directly represented and managed at the language level. For this reason, taking inspiration from PLP, we developed the DISPONTE semantics, discussed in Chapter 12. In DISPONTE, axioms are labeled with numeric parameters representing their probability. However these parameters are difficult to set for humans. On the other hand, data is usually available that can be leveraged for tuning them. We are thus interested in systems that automatically learn the probability values starting from the information available in the KB.

However, despite the adoption of the Semantic Web, knowledge bases are costly to manually update and so many are incomplete or incorrect. Moreover, a well specified KB can speed up the process of inference by avoiding false

information, useless subsumptions, etc. Therefore, there is a need of algorithms able to correct and/or improve the quality of the information modeled in KBs. In Chapter 22 we present EDGE, a system for learning the parameters of probabilistic KBs. EDGE is based on EMBLEM [13], a PLP learning system developed for learning the parameters for probabilistic logic programs under the distribution semantics. This can aid the creation of better KBs, but if we want to correct or improve information we need also algorithms able to learn the structure. The field of DLs structure learning is relatively recent and not many such algorithms exist. In Chapter 23 we present LEAP, an algorithm that combines parameter and structure learning in order to induce DISPONTE KBs.

In the last few years, the amount of data to process has exponentially increased. This led to the need of algorithm able to scale to large datasets. One of the possible solution to allowing that is the exploitation of parallelized and distributed approaches exploiting clusters and clouds. In Chaper 24 we show how EDGE and LEAP have been extended to cope with this necessity.

The tests described in Chapter 26 demonstrate the effectiveness of our approaches.

Chapter 22

EDGE: Parameter Learning

In this chapter, we present a supervised machine learning approach, implemented in the system EDGE ("Em over bDds for description loGics paramEter learning"). It is based on the algorithm EMBLEM [13] which is adapted to the case of probabilistic DLs under the DISPONTE semantics, thus it learns the parameters of DLs following the DISPONTE semantics from the information available in the domain. EDGE takes as input a KB and a number of examples of instances and non-instances of concepts that represent the queries. For each query, it generates the BDD encoding its explanations from the theory by means of BUNDLE (Chapter 15). Queries are divided into positive and negative examples: positive examples represent information that we regard as true and for which we would like to get high probability, while negative examples represent information that we regard as false and for which we would like to get low probability. The parameters are then tuned using an Expectation-Maximization (EM) algorithm [36] in which the required expectations are computed directly on the BDDs in an efficient way.

In the following, Section 22.1 describes in detail the EM algorithm used by EDGE. After that, Section 22.2 presents the learning algorithm.

22.1 Expectation Maximization Algorithm

EM [36] is an iterative algorithm in which two steps, called *expectation* and *maximization*, are repeated until the log-likelihood (LL) of the examples reaches a local maximum. At each iteration, the log-likelihood of the example in-

creases, i.e., the probability of positive examples increases and the one of negative examples decreases. The EM algorithm is guaranteed to find a local maximum, which however may not be the global maximum.

Given the examples in the form of BDDs, let us now present the formulas for the expectation and maximization phases:

- Expectation: for each query Q, EDGE computes $\mathbf{E}[c_{i0}|Q]$ and $\mathbf{E}[c_{i1}|Q]$ for all axioms F_i where c_{ix} is the number of times 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

  $$\mathbf{E}[c_{ix}] = \sum_Q \mathbf{E}[c_{ix}|Q]$$

- Maximization: EDGE computes p_i for all axioms E_i:

  $$p_i = \frac{\mathbf{E}[c_{i1}]}{\mathbf{E}[c_{i0}] + \mathbf{E}[c_{i1}]}$$

$P(X_i = x|Q)$ is given by $\frac{P(X_i=x,Q)}{P(Q)}$. Suppose for the moment that the BDD has been built without simplifying it with the deletion rule, i.e., each path from the root to the leaves contains one node for every variable. Then

$$P(X_i = x, Q) = \sum_{\rho \in R(Q)} P(X_i = x|\rho) \prod_{d \in \rho} p(d)$$

where ρ is a path, $R(Q)$ is the set of paths in the BDD for query Q that lead to a 1 leaf, $P(X_i = x|\rho) = 1$ if $X_i = x$ is in ρ and 0 otherwise, 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 $p(d) = 1 - p_i$. $P(X_i = x, Q)$ can be rewritten as

$$P(X_i = x, Q) \quad = \quad \sum_{\rho \in R(Q) \wedge (X_i = x) \in \rho} \prod_{d \in \rho} p(d)$$

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 BDD, $v(n)$ 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. So

$$\begin{aligned} P(X_i = x, Q) &= \sum_{n \in N(Q) \wedge v(n)=X_i} \sum_{\rho_n \in R_n(Q)} p_{ix} \prod_{d \in \rho_n} p(d) \sum_{\rho^n \in R^{child_x(n)}(Q)} \prod_{d \in \rho^n} p(d) \\ &= \sum_{n \in N(Q) \wedge v(n)=X_i} F(n) B(child_x(n)) p_{ix} \end{aligned}$$

where

$$F(n) = \sum_{\rho_n \in R_n(Q)} \prod_{d \in \rho_n} p(d)$$

is the *forward probability* [64], 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 the *backward probability* [64], the probability mass of paths from n to the 1 leaf. If *root* is the root of a tree for a query Q then $B(root) = P(Q)$ and $F(root) = 1$, while for terminal node $B(1) = 1$ and $B(0) = 0$.

The expression $F(n)B(child_x(n))p_{ix}$ represents the sum of the probabilities of all the paths passing through the x-edge of node n and is indicated with $e^x(n)$. Thus

$$P(X_i = x, Q) = \sum_{n \in N(Q) \wedge v(n)=X_i} e^x(n) \tag{22.1}$$

So $P(X_i = x, Q)$ is a sum of a contribution for each node associated with X_i, so all the nodes at the level of the X_i BDD variable. For the case of a fully simplified BDD, i.e., a diagram obtained by applying also the deletion

rule, Formula 22.1 is no longer valid since also paths where there is no node associated with X_i can contribute to $P(X_i = x, Q)$. Let $Del^x(X)$ be the set of nodes n such that the level of X is below that of n and is above that of $child_x(n)$, i.e., X is deleted between n and $child_x(n)$. These paths might have been obtained from a BDD having a node m associated to variable X_i that is a descendant of n along the 0-branch and whose outgoing edges both point to $child_0(n)$. The correction of formula (22.1) to take into account of this aspect is applied in the Expectation step.

In fact, suppose that a node n associated to variable Y has a level higher than variable X_i and suppose that $child_0(n)$ is associated to variable W that has a level lower than variable X_i. The nodes associated to variable X_i have been deleted from the paths from n to $child_0(n)$. One can imagine that the current BDD has been obtained from a BDD having a node m associated to variable X_i that is a descendant of n along the 0-branch and whose outgoing edges both point to $child_0(n)$. The original BDD can be re-obtained 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 $e^1(n)p_i$ for the paths passing through the 0-child and 1-child of m respectively.

Formally

$$
\begin{aligned}
P(X_{ij}=0,Q) &= \sum_{n\in N(Q)\wedge v(n)=X_{ij}} e^x(n) + \\
&\quad (1-p_i)\left(\sum_{n\in Del^0(X_{ij})} e^0(n) + \sum_{n\in Del^1(X_{ij})} e^1(n)\right) \\
P(X_{ij}=1,Q) &= \sum_{n\in N(Q)\wedge v(n)=X_{ij}} e^x(n) + \\
&\quad p_i\left(\sum_{n\in Del^0(X_{ij})} e^0(n) + \sum_{n\in Del^1(X_{ij})} e^1(n)\right)
\end{aligned}
$$

22.2 EDGE

EDGE performs supervised parameter learning. It is implemented in Java and is available at `https://sites.google.com/a/unife.it/ml/edge`. It takes as

Algorithm 8 Function EDGE: learning parameters of a (probabilistic) KB $\mathcal{K}$ given positive (E^+) and negative (E^-) examples.

```
1: function EDGE(K, E+, E−, ϵ, δ, NL, TL)
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:     Output: the final LL
10:    Output: probabilities p_i of the probabilistic axioms
11:    Build BDDs ▷ BUNDLE builds all the BDDs according to the limits NL and TL
12:    LL = −inf
13:    repeat
14:        LL_0 = LL
15:        LL = EXPECTATION(BDDs)
16:        MAXIMIZATION
17:    until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
18:    return (LL, p_i)
19: end function
```

input a DL theory, a number of *positive* examples (set E^+) and a number of *negative* examples (set E^-).

EDGE's main procedure, shown in Algorithm 8, first computes, for each example, the BDD encoding its explanations using BUNDLE [line 11]. A limit on the maximum number of explanations to be found (NL) or a time limit for the search for explanations (TL) can be optionally set. For a positive example of the form $a : C$, EDGE looks for the explanations of $a : C$ and encodes them in a BDD. For negative examples of the form $a : \neg C$, EDGE first looks for the explanations of $a : \neg C$, if one or more are found it encodes them into a BDD, otherwise it computes the explanations of $a : C$, builds the BDD and then negates it with the `NOT` BDD operator.

Then EDGE enters in the EM cycle in which the procedures EXPECTATION and MAXIMIZATION are repeatedly called [lines 15-16]. The first one returns the log likelihood LL of the data that is used in the stopping criterion [line 17]: EDGE stops when the difference between the LL of the current iteration and the one of the previous iteration (LL_0) drops below a threshold ϵ or when this difference is below a fraction δ of LL. Finally, EDGE returns LL and the probabilities p_i of the probabilistic axioms.

Algorithm 9 Function EXPECTATION

```
 1: function EXPECTATION(BDDs)
 2:     Input: the set of BDDs for the queries BDDs
 3:     Output: the computed LL
 4:     LL = 0
 5:     for all i ∈ Axioms do
 6:         E[c_i0] = E[c_i1] = 0
 7:     end for
 8:     for all BDD ∈ BDDs do
 9:         for all i ∈ Axioms do
10:             η^0(i) = η^1(i) = 0
11:         end for
12:         for all variables X do
13:             ς(X) = 0
14:         end for
15:         GETFORWARD(root(BDD))
16:         Prob=GETBACKWARD(root(BDD))
17:         T = 0
18:         for l = 1 to levels(BDD) do
19:             Let X_i be the variable associated with level l
20:             T = T + ς(X_i)
21:             η^0(i) = η^0(i) + T · (1 − p_i)
22:             η^1(i) = η^1(i) + T · p_i
23:         end for
24:         for all i ∈ Axioms do
25:             E[c_i0] = E[c_i0] + η^0(i)/Prob
26:             E[c_i1] = E[c_i1] + η^1(i)/Prob
27:         end for
28:         LL = LL + log(Prob)
29:     end for
30:     return LL
31: end function
```

Function EXPECTATION (Algorithm 9) This function takes as input a list of BDDs, one for each example Q, and computes the expectations $\mathbf{E}[c_{i0}|Q]$ and $\mathbf{E}[c_{i1}|Q]$ for all axioms E_i directly over the BDDs. Then it sums up the contributions of all examples $\mathbf{E}[c_{ix}]$.

In Algorithm 9 we use $\eta^x(i)$ to indicate $P(X_i = x, Q)$. EXPECTATION first calls GETFORWARD and GETBACKWARD [line 15-16] that compute the forward and the backward probability of nodes and $\eta^x(i)$ for non-deleted paths only. These are the paths that have not been deleted when building the BDDs. Then it updates $\eta^x(i)$ to take into account deleted paths, using the array ς. The expectations are updated in this way: for each axiom E_i, $\mathbf{E}[c_{ix}] = \mathbf{E}[c_{ix}] + \eta^x(i)/P(Q)$, where $P(Q)$ is the backward probability of the root [lines 24-27].

Algorithm 10 Procedure GETFORWARD: computation of the forward probability $F(n)$ in all BDD nodes n.

```
 1: procedure GetForward(root)
 2:     Input: the root node root of a BDD
 3:     F(root) = 1
 4:     F(n) = 0 for all nodes
 5:     for l = 1 to levels do
 6:         Nodes(l) = ∅
 7:     end for
 8:     Nodes(1) = {root}
 9:     for l = 1 to levels do
10:         for all node ∈ Nodes(l) do
11:             Let X_i be v(node), the variable associated with node
12:             if child_0(node) is not terminal then
13:                 F(child_0(node)) = F(child_0(node)) + F(node) · (1 − p_i)
14:                 Add child_0(node) to Nodes(level(child_0(node)))    ▷ level(node) returns
    node's level
15:             end if
16:             if child_1(node) is not terminal then
17:                 F(child_1(node)) = F(child_1(node)) + F(node) · p_i
18:                 Add child_1(node) to Nodes(level(child_1(node)))
19:             end if
20:         end for
21:     end for
22: end procedure
```

Procedure GETFORWARD, shown in Algorithm 10, initializes table F, where $F(n) = 1$ when $n = root$ and 0 otherwise [lines 3-4]. It also initializes table $Nodes$ [lines 5-7] that associates levels of the BDD with the nodes they contain. Then, it traverses the diagram one level at a time starting from the root level and for each node n it computes its contribution to the forward probabilities of its children [lines 9-19].

Function GETBACKWARD, shown in Algorithm 11, computes the backward probability of nodes by traversing recursively the tree from the leaves to the root. When the calls of GETBACKWARD for both children of a node n return [lines 8-9], we have all the information that is needed to compute the e^x values and the value of $\eta^x(i)$ for non-deleted paths [lines 10-13]. Array ς stores, for every level-variable l, an algebraic sum of $e^x(n)$: those for nodes in upper levels that do not have a descendant in level l minus those for nodes in upper levels that have a descendant in level l. In this way it is possible to add the contributions of the deleted paths by starting from the root level and accumulating $\varsigma(l)$ for the various levels in a variable T: an $e^x(n)$ value which

Algorithm 11 Procedure GETBACKWARD: computation of the backward probability, updating of η and of ς

```
 1: function GetBackward(node)
 2:     Input: a BDD node node
 3:     Output: the backward probability of node
 4:     if node is a terminal then
 5:         return value(node)
 6:     else
 7:         Let X_i be v(node)
 8:         B(child_0(node)) =GetBackward(child_0(node))
 9:         B(child_1(node)) =GetBackward(child_1(node))
10:         e^0(node) = F(node) · B(child_0(node)) · (1 − p_i)
11:         e^1(node) = F(node) · B(child_1(node)) · p_i
12:         η^0(i) = η^0(i) + e^0(node)
13:         η^1(i) = η^1(i) + e^1(node)
14:         VSucc = succ(v(node)) ▷ succ(X) returns the variable following X in the order
15:         ς(VSucc) = ς(VSucc) + e^0(node) + e^1(node)
16:         ς(v(child_0(node))) = ς(v(child_0(node))) − e^0(node)
17:         ς(v(child_1(node))) = ς(v(child_1(node))) − e^1(node)
18:         return B(child_0(node)) · (1 − p_i) + B(child_1(node)) · p_i
19:     end if
20: end function
```

is added to the accumulator T for level l means that n is an ancestor for nodes in this level. When the x-branch from n reaches a node in a level $l' \leq l$, $e^x(n)$ is subtracted from the accumulator, as it is not relative to a deleted node on the path anymore [lines 14-18].

Computing the forward and the backward probabilities of BDD nodes requires two traversals of the graph, so the cost is linear in the number of nodes.

Example 18. *Suppose you have the program of Example 9 and you have the single example kevin : NatureLover. The BDD of Figure 14.1 (also shown in Figure 22.1) is built and passed to* EXPECTATION *in the form of a pointer to its root node* n_1*. After initializing the* η *counters to 0,* GETFORWARD *is called with argument* n_1*. Table* F *for* n_1 *is set to 1 since this is the root. Then* F *is computed for the 0-child,* n_2*, as* $0 + 1 \cdot 0.6 = 0.6$ *and* n_2 *is added to* $Nodes(2)$*, the set of nodes for the second level. Then* F *is computed for the 1-child,* n_3*, as* $0 + 1 \cdot 0.4 = 0.4$*, and* n_3 *is added to* $Nodes(3)$*. In the next iteration, level 2 is considered and node* n_2 *is fetched from* $Nodes(2)$*. The 0-child is a terminal so it is skipped, while the 1-child is* n_3 *and its* F *value is updated as* $0.4 + 0.6 \cdot 0.3 = 0.58$*. In the third iteration, node* n_3 *is fetched but since its children are leaves,* F *is not updated. The resulting forward probabilities are*

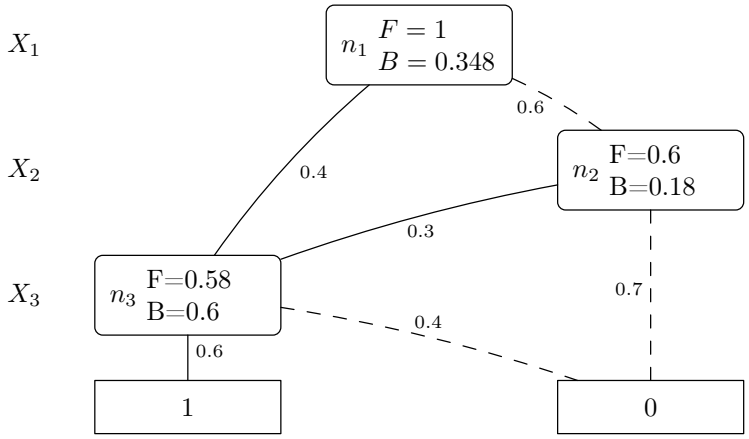


Figure 22.1: Forward and backward probabilities (indicated respectively by F and B) of each node of the BDD of Example 9.

shown in Figure 22.1.

Then GetBackward *is called on* n_1 *which is not a terminal node, hence the function calls* GetBackward(n_2) *that in turn calls* GetBackward(0). *The latter call returns 0 because it is a terminal node. After that, the function calls* GetBackward(n_3) *on node* n_3 *whose children are both terminal:* GetBackward(1) *returns 1 and* GetBackward(0) *returns 0. Then* GetBackward(n_3) *computes* $e^0(n_3)$ *and* $e^1(n_3)$ *in the following way:*

$e^0(n_3) = F(n_3) \cdot B(0) \cdot (1 - p_3) = 0.58 \cdot 0 \cdot 0.4 = 0$

$e^1(n_3) = F(n_3) \cdot B(1) \cdot (\pi_{21}) = 0.58 \cdot 1 \cdot 0.6 = 0.348$

where $B(n)$ *and* $F(n)$ *are respectively the backward and forward probabilities of node* n. *Now the counters for clause* C_3, *are updated:*

$\eta^0(3) = 0$

$\eta^1(3) = 0.348$

while we do not show the update of ς *since its value for the level of the leaves is not used afterwards.* GetBackward(n_3) *now returns the backward probability of* n_3 $B(n_3) = 0{\cdot}0.4{+}1{\cdot}0.6 = 0.6$. GetBackward($n_2$) *can proceed to compute*

$e^0(n_2) = F(n_2) \cdot B(0) \cdot (1 - p_2) = 0.6 \cdot 0.0 \cdot 0.7 = 0$

$e^1(n_2) = F(n_2) \cdot B(n_3) \cdot (p_2) = 0.6 \cdot 0.6 \cdot 0.3 = 0.216$

and $\eta^0(2) = 0$, $\eta^1(2) = 0.216$. *The variable following* X_2 *is* X_3 *so* $\varsigma(X_3) = e^0(n_2) + e^1(n_2) = 0 + 0.216 = 0.216$. *Since* X_2 *is also associated to the 1-child* n_3, *then* $\varsigma(X_3) = \varsigma(X_3) - e^1(n_2) = 0$. *The 0-child is a leaf so we do not show*

the update of ς .

GetBackward(n_2) *then returns* $B(n_2) = 0 \cdot 0.7 + 0.6 \cdot 0.3 = 0.18$ *to* GetBackward(n_1) *that calls* GetBackward(n_3) *that computes* $e^0(n_1)$ *and* $e^1(n_1)$ *as*

$e^0(n_1) = F(n_1) \cdot B(n_2) \cdot (1 - p_1) = 1 \cdot 0.18 \cdot 0.6 = 0.108$

$e^1(n_1) = F(n_1) \cdot B(n_3) \cdot (p_1) = 1 \cdot 0.6 \cdot 0.4 = 0.24$

and updates the η *counters as* $\eta^0(1) = 0.108$, $\eta^1(1) = 0.24$.

Finally ς *is updated:*

$\varsigma(X_2) = e^0(n_1) + e^1(n_1) = 0.108 + 0.24 = 0.348$

$\varsigma(X_2) = \varsigma(X_2) - e^0(n_1) = 0.24$

$\varsigma(X_3) = \varsigma(X_3) - e^1(n_1) = -0.24$

GetBackward(n_1) *returns* $B(n_1) = 0.18 \cdot 0.6 + 0.6 \cdot 0.4 = 0.348$ *to* Expectation, *which adds the contribution of deleted nodes by cycling over the BDD levels and updating* T. *Initially* T *is set to 0, then, for variable* X_1, T *is updated to* $T = \varsigma(X_1) = 0$ *which implies no modification of* $\eta^0(1)$ *and* $\eta^1(1)$. *For variable* X_2, T *is updated to* $T = 0 + \varsigma(X_2) = 0.24$ *and table* η *is modified as*

$\eta^0(2) = 0 + 0.24 \cdot 0.7 = 0.168$

$\eta^1(2) = 0.216 + 0.24 \cdot 0.3 = 0.288$

For variable X_3, T *becomes* $0.24 + \varsigma(X_3) = 0$ *so* $\eta^0(3)$ *and* $\eta^1(3)$ *are not updated. At this point the expected counts for the three axioms can be computed:*

$\mathbf{E}[c_{10}] = 0 + 0.108/0.348 = 0.310$

$\mathbf{E}[c_{11}] = 0 + 0.24/0.348 = 0.690$

$\mathbf{E}[c_{20}] = 0 + 0.168/0.348 = 0.483$

$\mathbf{E}[c_{21}] = 0 + 0.288/0.348 = 0.828$

$\mathbf{E}[c_{30}] = 0 + 0/0.348 = 0$

$\mathbf{E}[c_{31}] = 0 + 0.348/0.348 = 1$

Procedure Maximization (Algorithm 12) This procedure computes the parameters values p_i for the next EM iteration by relative frequency using the values of the expected counts.

Example 19 (Example 18 cont.)**.** *The expected counts have been computed by the expectation step, thus EDGE can execute the maximization step to tune the parameters of the probabilistic axioms.*

Algorithm 12 Procedure MAXIMIZATION

```
1: procedure MAXIMIZATION
2:     for all i ∈ Axioms do
3:         p_i = E[c_i1]/(E[c_i0] + E[c_i1])
4:     end for
5: end procedure
```

$$p_1 = \mathbf{E}[c_{11}]/(\mathbf{E}[c_{10}] + \mathbf{E}[c_{11}]) = 0.690/(0.310 + 0.690) = 0.690$$
$$p_2 = \mathbf{E}[c_{21}]/(\mathbf{E}[c_{20}] + \mathbf{E}[c_{21}]) = 0.828/(0.483 + 0.828) = 0.632$$
$$p_3 = \mathbf{E}[c_{31}]/(\mathbf{E}[c_{30}] + \mathbf{E}[c_{31}]) = 1/(0 + 1) = 1$$

We remind that previous parameter values were $p_1 = 0.4$, $p_2 = 0.3$ *and* $p_3 = 0.6$. *As expected, since we have only a single positive example whose explanations contain all the probabilistic axioms, their parameters increase.*

Chapter 23

LEAP: Structure Learning

In this chapter we present LEAP ("LEArning Probabilistic description logics"), an algorithm able to learn both parameters and structure of DISPONTE KBs. LEAP combines two algorithms, CELOE [83] and EDGE, as shown in Figure 23.1. 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.

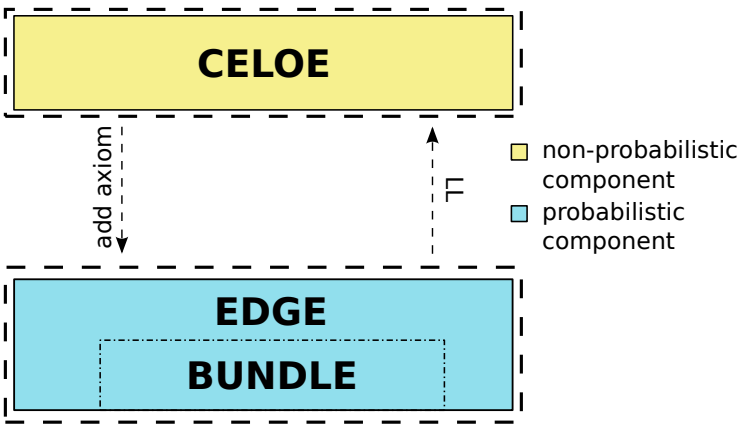


Figure 23.1: LEAP's architecture.

LEAP is written in Java and is available at `https://sites.google.com/a/unife.it/ml/leap`.

In the following, Section 23.1 presents CELOE and gives an overview of class expression learning for DLs, while Section 23.2 shows LEAP's code and describes the algorithm.

23.1 CELOE

CELOE [83] stands for “Class Expression Learning for Ontology Engineering" and is available in the Java open-source framework DL-Learner[1] for OWL and DLs.

Suppose you are given a knowledge base $\mathcal{K}$ and that we want to learn a formal description for a class `Target` which has (inferred or asserted) instances in $\mathcal{K}$. CELOE takes as input a target class and a set of positive and negative (or only positive) examples (i.e. *individuals*) and solves one of Class Learning Problem or Learning from Examples Problem.

If `Target` is already described by a class expression C, i.e., there are already axioms such as $\texttt{Target} \sqsubseteq C$ or $\texttt{Target} \equiv C$ in $\mathcal{K}$, it is possible to learn a description for `Target` by refining C or by relearning it from scratch, as stated in Definition 4.

Definition 4 (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* $R_{\mathcal{K}}(\texttt{Target}) = R_{\mathcal{K}}(C)$*.*

CELOE creates a set of n class expressions C_i $(1 \leq i \leq n)$ and serts them according to a heuristic. Such expressions are candidates for adding axioms of the form $\texttt{Target} \equiv C_i$ or $\texttt{Target} \sqsubseteq C_i$.

Otherwise, if a set of positive and negative examples or a set of positive only examples is available, CELOE can exploit them to solve a problem of learning from examples, as described in Definition 5.

Definition 5 (Learning from Examples Problem)**.** *Given:*

- *a concept name* `Target`*;*
- *a knowledge base* $\mathcal{K}$ *not containing* `Target`*;*
- *a space of possible concepts* $\mathcal{C}$*;*
- *a set of positive examples* E^+ *with elements of the form* a : `Target` $(a \in \mathbf{I})$*;*

[1] http://dl-learner.org/Projects/DLLearner

- *a set of negative examples* E^- *with elements of the form* a : `Target` ($a \in \mathbf{I}$)*;*

Find a concept expression $C \in \mathcal{C}$ *such that:*

- `Target` *does not occur in* C *(acyclic definition);*
- $\forall e^+ \in E^+$*, the concept* C covers *the example* e^+*, i.e.,* $\mathcal{K} \cup \{\texttt{Target} \equiv C\} \models e^+$*;*
- $\forall e^- \in E^-$*, the concept* C *does not* cover *the example* e^-*, i.e.,* $\mathcal{K} \cup \{\texttt{Target} \equiv C\} \not\models e^-$.

Here, if both sets E^+ and E^- of individuals are given the problem takes the name of *Positive and Negative Examples Learning Problem*, while if only the set E^+ is available it is called *Positive Examples Learning Problem*.

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.

CELOE is a top-down algorithm that starts from the $\top$ class expression and uses the $\mathcal{ALCQ}$ refinement operator defined in [84]. Each generated class expression is evaluated using one of five available heuristics, whose value is used to guide the search. All these heuristics need a set of examples in order to be computed; in the case the algorithm is solving a class learning problem where no examples are given, we can consider as positive examples the existing instances (inferred or asserted) of the target class and the remaining instances in the KB as negative examples.

Performing instance retrieval $R_\mathcal{K}$ can be very expensive for large ontologies. In order to make CELOE scalable, three performance optimizations are provided:

Reduction of instance checks: it exploits background knowledge in order to reduce the number of considered individuals. If we know that class `Target` has a super class A, top-down search looks for individuals belonging to A instead of $\top$. In this way the number of negative examples is lower.

Approximate and closed world reasoning: it consists of using reasoners specifically designed for performing a high number of instance checks in the lowest time by partially following 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 [2]. See [83] and [2] for further details.

23.2 LEAP

In order to learn the structure of a KB, LEAP first finds good candidate subsumption axioms by means of CELOE, then it performs a greedy search in the space of theories.

LEAP main procedure, shown in Algorithm 13, 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 NL and the time limit TLE for the computation of the BDDs for each example for EDGE[2].

In the first phase, a set of class expressions is generated by using CELOE [line 11], then the sets of positive (P_I) and negative (N_I) individuals are extracted depending on the learning problem LP_{type}:

- if LP_{type} = *Positive and Negative Examples Learning Problem*, then no extraction is necessary since a set of positive and negative individuals has been given;
- if LP_{type} = *Positive Examples Learning Problem*), then only the set of negative examples must be created. In this case this set will contain all the individuals of $\mathcal{K}$ except the positive ones;
- if LP_{type} = *Class Learning Problem* (cf. Definition 4) where only the target class has been given, then we consider the existing instances (inferred or asserted) of the target class as positive individuals and the remaining instances as negative individuals.

[2]Default values are: $NC = 10$, $TLC = 10$ seconds and $NE = TLE = \infty$

Algorithm 13 Function LEAP.

```
1: function LEAP(𝒦, LP_type, NC, TLC, ϵ, δ, NL, TLE)
2:     Input: a knowledge base 𝒦
3:     Input: the type LP_type of learning problem
4:     Input: the maximum number of class expressions to find NC
5:     Input: the time limit for the inference for CELOE TLC
6:     Input: a threshold ϵ for the difference between LLs
7:     Input: a threshold δ for the fraction of the difference between LLs
8:     Input: the maximum number of explanations to find for each example NL
9:     Input: the time limit for the inference process for each example TLE
10:    Output: the learned knowledge base 𝒦
11:    ClassExpressions = up to NC or until TLC is reached   ▷ generated by CELOE
12:    (P_I, N_I) = ExtractIndividuals(LP_type)   ▷ LP_type: specifies how to extract (P_I, N_I)
13:    for all ind ∈ P_I do   ▷ P_I: set of positive individuals
14:        Add ind : Target to E+   ▷ E+: set of positive examples
15:    end for
16:    for all ind ∈ N_I do   ▷ N_I: set of negative individuals
17:        Add ind : Target to E−   ▷ E−: set of negative examples
18:    end for
19:    (LL_0, 𝒦) = EDGE(𝒦, E+, E−, ϵ, δ, NL, TLE)
20:    for all CE ∈ ClassExpressions do
21:        Axiom = p :: CE ⊑ Target
22:        𝒦' = 𝒦 ∪ {Axiom}
23:        (LL, 𝒦') = EDGE(𝒦', E+, E−, ϵ, δ, NL, TLE)
24:        if LL > LL_0 then
25:            𝒦 = 𝒦'
26:            LL_0 = LL
27:        end if
28:    end for
29:    return 𝒦
30: end function
```

After the extraction, the *assertional* axioms, which represent the examples (i.e. queries) for EDGE, are created [lines 13-18]. Then EDGE is applied to the KB to compute the initial value of the parameters and of the log-likelihood LL [line 19].

In the second phase, LEAP performs a greedy search in the space of theories [lines 20-28]. For each element CE of the class expressions set, one probabilistic subsumption axiom at a time of the form $p :: CE \sqsubseteq \texttt{Target}$ is added to the ontology $\mathcal{K}$ where p is either a random probabilistic value or the accuracy returned by CELOE. After each addition, EDGE is run on the extended theory to compute the log-likelihood of the data LL and the updated parameters [line 23]. If LL is better than the current best LL_0, the new axiom is kept in the

knowledge base, otherwise the new axiom is discarded [lines 24-27]. The final theory, obtained from the union of the initial ontology and the probabilistic subsumption axioms learned, is returned to the user.

Chapter 24

Distributed Learning

In the last few years, the pervasiveness of Internet, the availability of sensor data, the dramatically increasing storage and computational capabilities provided the opportunity to gather exponentially increasing sets of data, the so-called *Big Data*. The Semantic Web paved the way to the creation of Big Data in the form of *Open Linked Data* where information is often distributed on many different nodes. In previous chapters we presented two approaches for learning from data. The main issue is given by the running time, the algorithms may take hours on datasets of the order of MBs, depending on many aspects such as the type of examples, the level of complexity of the KBs, etc. In a field where the amount of data to be processed is large, it is of foremost importance to develop approaches with the ability to scale, for taking into account more data. One solution is to distribute algorithms exploiting modern high performance computing infrastructures, such as clusters and clouds.

Between 2004 and 2008 Google, guided by its needs to process large amounts of raw data, presented and then extended a framework, called MapReduce [35], for handling data of the order of Terabytes. In this model the work load is distributed among mapper and reducer workers which execute *map* and *reduce* operations that aggregate data. MapReduce was inspired by the Message Passing Interface (MPI) standard, developed in the 1995, which has *reduce* and *scatter* operations.

In order to adapt our learning algorithms to the management of Big Data, we implemented a parallelized version of EDGE, called EDGE$^{\mathrm{MR}}$. It was then integrated into LEAP, leading to the implementation of LEAP$^{\mathrm{MR}}$. Various

MapReduce frameworks are available, such as Hadoop[1]. However, standard MapReduce frameworks require purely functional operations, which may not be optimal for our algorithms. Hence we chose not to use any framework and to implement a basic MapReduce approach for EDGE^{MR} based on MPI.

In the following, in Section 24.1 we give an introduction on MapReduce and briefly discuss why it may not be optimal for our purposes while in Section 24.2 we present the MPI standard. Finally, Section 24.3 describes EDGE^{MR} and Section 24.4 discusses LEAP^{MR}.

24.1 Map Reduce Approach

The name MapReduce [35] was inspired by two standard functions of many functional programming languages such as LISP, namely *map()* and *reduce()*. MapReduce adds a variety of optimizations to make the functions scalable and fault-tolerant. These two aspects are extremely important since MapReduce is based on a distributed paradigm.

A MapReduce program is composed of two main steps:

Map Step: the data is taken in input, divided into chunks and distributed among workers by a *master* process. In this phase, each worker processes input data using a function *map()* defined by the user. The *map()* function takes on pair $(key, value)$, produces one or more $(stack\_key, value')$ pairs and saves them to a shared file, usually in a distributed file system, in a location known by all the nodes involved in the process.

Reduce Step: it is divided in two sub-steps: the *shuffle step*, where the master collects the files created by the mappers and combines pairs with the same key. The result is a set of pairs $(stack\_key, list\_of\_values)$, which is the input for the *reduce step*. The *reduce step* executes a *reduce()* function, also written by the user, which produces one or more output results - typically $(stack\_key, agg\_value)$. The output is saved to a shared file.

In many applications, the map and reduce phases are iteratively called, in these cases the output of the reduce function is used as input for a new execution of the map function.

[1] `https://hadoop.apache.org/`

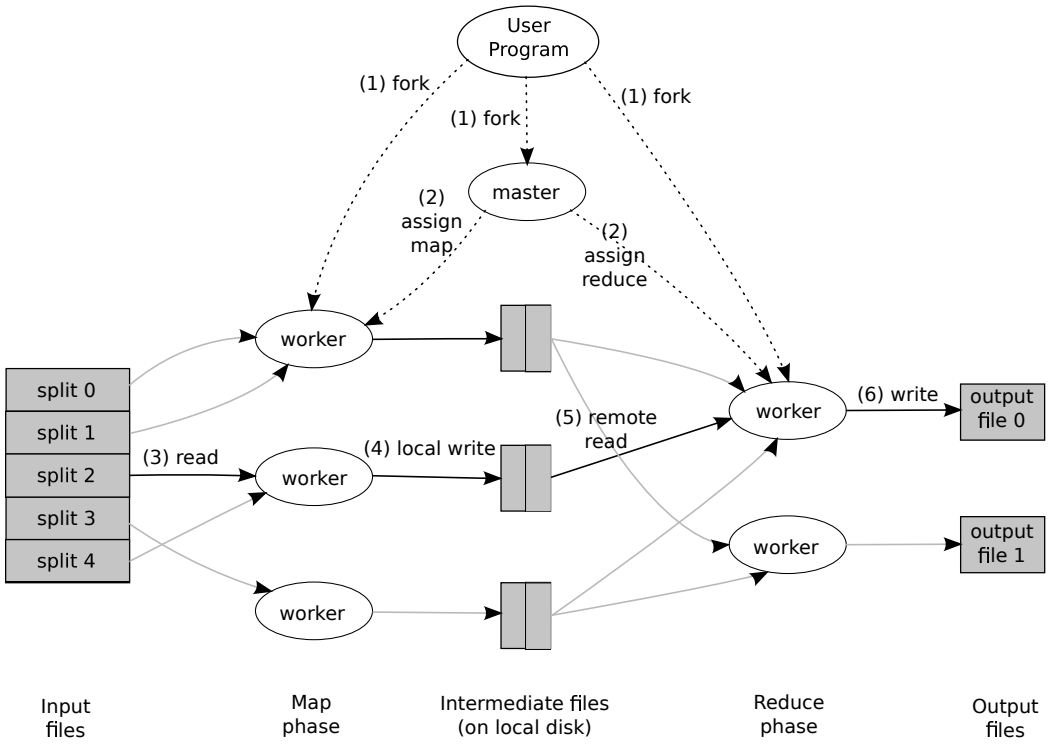


Figure 24.1: Overall flow of a MapReduce operation (from [35]).

Figure 24.1 shows the general flow of a single iteration of MapReduce. Usually, for each step shown in the figure, several different parameters can be set to improve performances. First, all workers are initialized [steps 1-2] and the input files are split. Here, the partitioning of the output is usually done using hashing, but different partitioning function can be used. Then each worker (mapper) reads its input [step 3], elaborates it and produces a file containing the results [step 4]. Afterwards, the reducers read these files and aggregate the pairs, producing the output files [step 6].

Nowadays there are many different implementations of MapReduce. One of the most used is Hadoop, an open-source framework implemented by the Apache Software Foundation. Usually, MapReduce frameworks exploit HDFS as distributed file systems.

In our algorithm, we would like to keep in main memory information such as the BDDs of the examples because building a BDD from the set of explanations is very expensive (cf. Chapter 18). We think that keeping in memory such information can improve significantly the running time of the algorithm. For

these reasons, we decided not to exploit MapReduce frameworks, but we used a simpler implementation of a MapReduce algorithm using MPI which is at the basis of most of the available frameworks.

24.2 The Message Passing Interface Standard

The MPI protocol was standardized in 1994 after four years of work by the MPI Forum[2], a group of 80 people from 40 organizations, universities, governments and industries. In the following years, the MPI Forum updated the protocol. Today, the last version is MPI-3, approved in 2012.

The MPI standard defines the syntax and semantics of a core library of routines for implementing distribute programs using mainly Fortran or C. One of the most used implementation of the MPI protocol is OpenMPI[3], which permits the use of MPI also with Java by providing an interface to the native library (JNI).

MPI processes can be assigned to a different CPU or a different machine. The assignment is done at run-time, when a first initialization of all the processes assigns a *rank* to each of them and creates *communication worlds*. The rank is a number greater or equal to 0 used to identify every process. Typically, the rank 0 process is the master. Communication worlds are used to group processes that can communicate with each other. Every time a process sends a message, it has to specify which *communicator* should be used for the communication, i.e., which group of processes can receive the message. Each process is assigned to the MPI_COMM_WORLD communicator, which groups all the processes, and can be assigned to other different application specific communication worlds. Communications can be of many types such as synchronous or asynchronous, one-to-one, one-to-many, many-to-many or broadcast.

MPI also defines many functions for synchronizing nodes, dividing and sending data (scattering), aggregating and combining (partial) results (gathering and reducing), recovering information from the network, etc.

[2] `http://www.mpi-forum.org/`
[3] `http://www.open-mpi.org/`

24.3 EDGE^{MR}

EDGE^{MR} uses MPI to distribute the computational load of EDGE following an approach similar to MapReduce. As discussed in Section 24.1, the processes of EDGE^{MR} are not purely functional, as required by standard MapReduce, because they have to retain in main memory BDDs during the whole execution.

Like most MapReduce frameworks, the EDGE^{MR} architecture follows a master-slave model. For the communication between the master and the slaves, OpenMPI is adopted. 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.

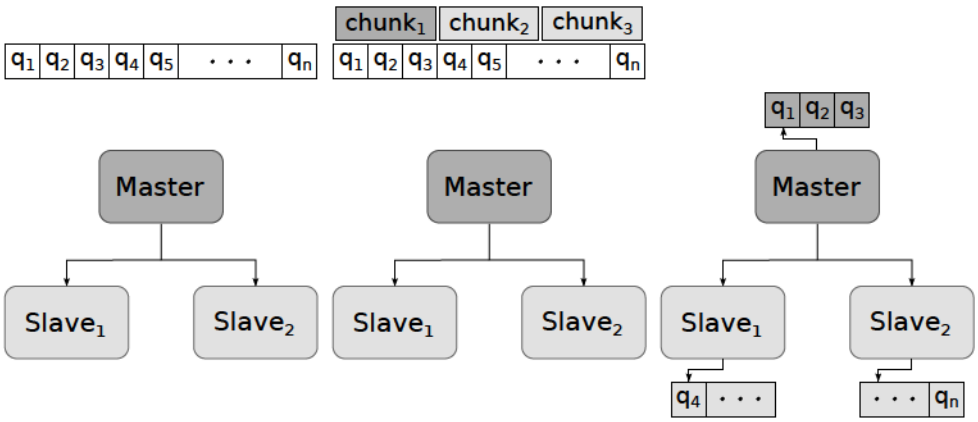
Initialization During this phase the data is replicated and a process is created on each machine. Then each process parses its copy of the probabilistic KB and stores it in main memory. The master, in addition, parses files containing positive and negative examples (the queries).

Query resolution The master divides the set of queries into chunks 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, called *single-step* and *dynamic*.

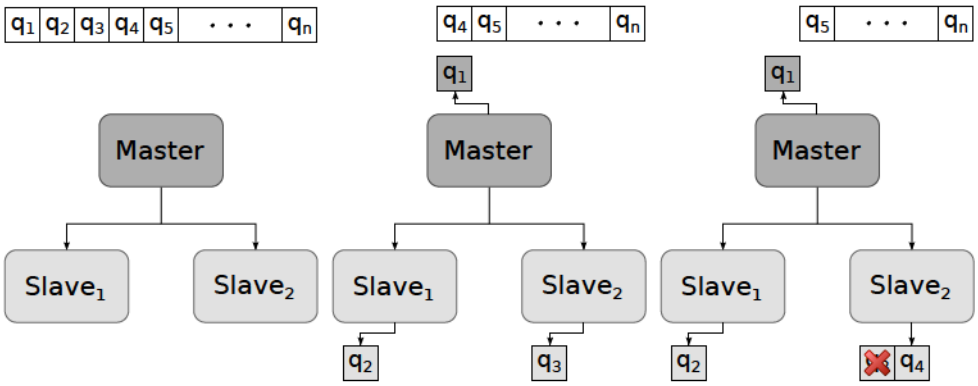
Expectation-Maximization Once all nodes have built the BDDs for their queries, EDGE^{MR} starts the Expectation-Maximization cycle. During the Expectation step all the workers traverse their BDDs and calculate their local array η. Then the master gathers all the ηs from the workers and aggregates them 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.

Scheduling Techniques In a distributed context, the scheduling strategy influences significantly the performances. We developed two scheduling strategies, *single-step scheduling* and *dynamic scheduling*. The scheduling technique is chosen during the initialization phase and affects only the generation of the

BDDs for the queries. The initialization and EM phases are independent of the chosen scheduling method.



(a) Single-step scheduling



(b) Dynamic scheduling

Figure 24.2: Scheduling techniques of EDGE^{MR}.

Single-step Scheduling If N is the number of the slaves, the master divides the queries into $N+1$ chunks, i.e. the number of slaves plus the master. Then the master starts $N+1$ threads, one building the BDDs 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 24.2a shows an example of single-step scheduling with two slaves.

Dynamic Scheduling This method 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. At first, each machine is assigned one example of queries in order. When a worker finishes handling the example, it takes the following. So if the master ends handling its example, it just picks the next one, while if a slave ends handling its example, it asks the master for another one. During this phase the master runs a listener thread that waits for slaves' requests of new examples. For each request, the listener starts a new thread that sends an example to the requesting slave (to improve the performances this is done through a thread pool). When all the BDDs for queries are built, EDGE$^{\text{MR}}$ starts the EM cycle. An example of dynamic scheduling with two slaves and a chunk dimension of one example is displayed in Fig. 24.2b.

MapReduce View From a MapReduce point of view, after the initialization phase, the map phase executes query resolution and the expectation step, while the reduce phase concerns only maximization. In particular

Map This phase is performed by every process. First *query resolution* is performed where each worker builds its private set of BDDs, then *expectation* is executed where each worker calculates its local η. The output pairs *(key, value)* contain an example identifier as *key* and the array η as *value*.

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. The master executes the *maximization step*, where it gathers all η arrays from the workers, sums them component wise, performs the maximization step and sends the newly updated parameters to the slaves.

The Map and Reduce phases implement the functions Expectation and Maximization 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 maintain in memory the generated BDDs for the whole execution of the EM cycle.

In EDGE$^{\text{MR}}$'s main procedure, shown in Algorithm 14, first each process reads the given input. Then the master, depending on the scheduling, sends the examples to the slaves 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 an example to the slaves at every request it receives. During this time, the slaves, depending on the scheduling type, receive the examples and build the corresponding BDDs [lines 40-49]. All the BDDs are built by BUNDLE according to the limits NL and TL. After that, 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 9) can be executed by all the workers [lines 32 and 52]. Finally, all workers enter in the maximization phase where the master collects all the values, executes the MAXIMIZATION procedure (Algorithm 12) and checks whether a new round of EM must be performed [line 33-37], while the slaves only wait for a signal from master which indicates whether to execute either EXPECTATION or stop.

24.4 LEAP$^{\text{MR}}$

LEAP$^{\text{MR}}$ is an evolution of the system LEAP presented in Section 23.2. While the latter exploits EDGE, the first was adapted to perform EDGE$^{\text{MR}}$.

Algorithm 15 shows LEAP$^{\text{MR}}$'s main procedure, where the highlighted lines are those which differ from the serial version of LEAP. It takes as input the knowledge base $\mathcal{K}$ and configuration settings for CELOE and EDGE$^{\text{MR}}$, then generates at most NC class expressions by exploiting CELOE and the sets of positive and negative examples which will be the queries for EDGE$^{\text{MR}}$ [lines 12-19]. A first execution of EDGE$^{\text{MR}}$ is applied to $\mathcal{K}$ to compute the initial value of the parameters and of the LL [line 20]. Then LEAP$^{\text{MR}}$ adds to $\mathcal{K}$ one probabilistic subsumption axiom at a time. After each addition, EDGE$^{\text{MR}}$ is performed on the extended KB to compute the LL of the data and the parameters [line 24]. If the LL is better than the current best, the new axiom is kept in the knowledge base and the parameters of the probabilistic axioms are updated, otherwise the learned axiom is removed from the ontology and

Algorithm 14 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
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
```

Algorithm 15 Function LEAP$^{\mathrm{MR}}$.

function LEAP$^{\mathrm{MR}}$($\mathcal{K}, LP_{type}, NC, TLC, \epsilon, \delta, NL, TLE, S$)
  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^+: 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
  $(LL, \mathcal{K}') = EDGE^{MR}(\mathcal{K}', E^+, E^-, \epsilon, \delta, NL, TLE, S)$ ▷ Call to $EDGE^{MR}$
  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

the previous parameters are restored. The final theory is obtained from the union of the initial ontology and the probabilistic axioms learned.

Chapter 25

Related Learning Systems

In this chapter, we discuss the approaches which are close to our systems.

A work that integrates parameters and structure learning for the probabilistic extension CR$\mathcal{ALC}$, is [90, 114, 91]. As reported in Section 13, CR$\mathcal{ALC}$ adopts an interpretation-based semantics and allows, besides $\mathcal{ALC}$ constructs, the probabilistic axioms $P(C|D) = \alpha$, meaning that for any element x of the domain, 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.

The acyclicity assumption in CR$\mathcal{ALC}$ enables to represent any KB $\mathcal{K}$ as a directed acyclic graph $G(\mathcal{K})$ which is a template for generating a ground graph given the domain in which each node represents an instantiated logical atom $C(a)$ or $R(a, b)$.

The algorithm of [91] learns parameters and structure of CR$\mathcal{ALC}$ knowledge bases. It starts from positive and negative examples for a single concept and from the general concept $\top$ in the root of the search tree. The space of possible concept definitions is explored by means of a revision operator in the style of Inductive Logic Programming. For a set of candidate definitions of a given length, their parameters are learned using an EM algorithm, since the ground graph contains unobserved variables, as in EDGE. In particular, 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 $P(C_i|Parents(C_i))$. Once the parameters for

the candidate definitions are learned, each definition is scored against the examples: the score is the product of the probability of the examples given the background terminology and the definition. The definition with the highest score is retained and the algorithm enters a new refinement iteration. The cycle ends when the difference between two best scores is below a threshold. We share the top-down procedure for building axioms (CELOE) but we exploit BDD structures to compute the expected counts for EM instead of resorting to inference in a graphical model.

The paper [95] 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) modeling 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 Open World Assumption) regarding concept-membership, reflecting in the adoption of different scoring functions and search strategies of the optimal network and parameters. Under one of these 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.

Another approach is presented in [151, 44], where the authors introduce an algorithm, called GoldMiner, that exploits Association Rules (ARs) for building ontologies. GoldMiner extracts information about individuals, named classes and roles using SPARQL queries. Then, starting from this 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 pair of individuals belongs. The first contains a row for each individual and a column for all named classes and classes of the form $\exists R.C$ for R a role and C a named class. The cells of the table contain 1 if the individual belongs to the class of the column and 0 otherwise. The second table contains a row for each pair of individuals and a column for each named role. The cells contain 1 if the pair of individuals belongs to the role in the column and 0 otherwise. Finally, the

APRIORI algorithm [1] is applied to each table in turn in order to find ARs. ARs are implications of the form $A \Rightarrow B$ where A and B are conjunctions of columns (and thus conjunctions of classes or roles). Each AR of the form $A \Rightarrow B$ can thus be converted to the axiom $A \sqsubseteq B$. So from the learned ARs a knowledge base can be obtained. Moreover, each AR $A \Rightarrow B$ is associated with a confidence that is the fraction of transactions that satisfy B among those that satisfy A. Thus the confidence 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 parameters learner EDGE is inspired by EMBLEM [13], an algorithm developed to learn the parameters of probabilistic logic programs under the distribution semantics, as seen at the beginning of Chapter 22. It shares with EDGE the use of EM algorithm and the exploitation of knowledge compilation, in particular BDDs, for computing the distribution of the hidden variables.

The structure learner LEAP is inspired by SLIPCOVER [14], an algorithm proposed for learning probabilistic logic programs based on the distribution semantics. LEAP shares with it the search strategy and the use of the log-likelihood of the data as the score of the learned theories. Like SLIPCOVER, it divides the search between learning promising axioms and building in a greedy way a theory (KB) whose parameters are optimized by relying on a parameter learning algorithm. A MapReduce approach was applied also to SLIPCOVER, developing SEMPRE. As for LEAP$^{\mathrm{MR}}$, MapReduce was implemented by directly exploiting MPI. In SEMPRE, mapper workers keep in memory data structures across MapReduce iterations and the reduce strategy is particularly simple, being realized by a single reducer receiving the output from all mapper jobs.

Chapter 26

Experiments

In order to test the performances of our learning systems, we performed several experiments, both on the quality of the results and on the improvement introduced by the application of distributed approaches. In the following, each section presents a different test. At the end, Section 26.6 discusses the results. Experiments presented in Sections 26.1 and 26.2 have been performed on a cluster of 64-bit Linux machines with 2 GB (max) memory allotted to Java per node where each node of this cluster has 2-cores Intel E6550 2.33 GHz CPUs. Experiments in Sections 26.3, 26.4 and 26.5 have been performed 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.

26.1 EDGE: Comparison with Association Rules

EDGE has been compared with Association Rules (ARs) over two real world datasets from the Linked Open Data cloud: EDU-UK[1], which contains information about school institutions in the United Kingdom, and an extract of DBPedia[2] [85], a knowledge base obtained by extracting structured data from Wikipedia. We took in consideration only ARs because in this test we focused only on the parameter learning problem.

In the experiments, we wanted to simulate the situation in which an expert provides the structure of the ontology together with information on a set of

[1] `http://education.data.gov.uk/`
[2] `http://dbpedia.org/`

individuals. The ontologies were obtained with GoldMiner using the following parameters for the APRIORI algorithm: 0.1 as the minimum support and 0.05 as the minimum confidence. We extracted 10,000 individuals and 5,545 axioms for EDU-UK and 7,200 individuals and 6,228 axioms for DBPedia. Then we learned ARs from the resulting transaction tables. Note that in this test we considered only the ARs that can be converted into subclass axioms.

Then we selected positive and negative examples. We first randomly chose individuals from the extracted ones. For each individual *ind* we identified two named classes: A, that is randomly selected among the classes to which *ind* belongs that do not have subclasses, and B, that is randomly selected from all the classes to which *ind* belongs. We add the resulting triple (ind, A, B) to a set P. Then, for each triple (ind, A, B) in P we added $ind : A$ to the ontology and $ind : B$ to the set of positive examples.

Negative examples were selected in the following way:

1. we randomly chose individuals from the extracted ones;
2. for each selected individual *ind*, we randomly chose:
   - a named class A from the list of classes to which *ind* belongs;
   - a named class B appearing in the ontology for which we do not know explicitly whether *ind* belongs to or not;

   we test the satisfiability of the query $ind : B$ w.r.t. the knowledge base that contains the axiom $ind : A$. If the query is satisfiable, we add (ind, A, B) to the set N.

Finally, for each triple (ind, A, B) in N we added $ind : A$ to the ontology and $ind : B$ to the set of negative examples.

We used a 5-fold cross validation to test the system: we partitioned the set of queries in five equally sized subsets and we performed five experiments in which we used four subsets for training and one for testing.

In the training phase, we ran EDGE on the ontology obtained by GoldMiner where we considered all the axioms as probabilistic. We randomly set the initial values of the parameters. EDGE, for handling 5,000 examples, took about 15,000 seconds in average for DBPedia, about 3 seconds per example, and about 173,000 seconds in average for EDU-UK, about 34 seconds

Table 26.1: Areas under the ROC and PR curves with standard deviation, execution times and p-value of a paired two-tailed t-test at the 5% significance level for EDGE and Association Rules.

Datasets		EDGE	ARs	p-value
EDU-UK	PR	0.9702 ± 0.0289	0.8804 ± 0.0165	0.0051
	ROC	0.9796 ± 0.0166	0.9158 ± 0.0171	0.0093
	Time (s)	173,528	10,490	
DBPedia	PR	0.9784 ± 0.0483	0.5916 ± 0.0999	0.0013
	ROC	0.9902 ± 0.0219	0.4346 ± 0.1319	0.0007
	Time (s)	14,883	578,420	

per example. Most of the runtime was spent in finding the explanations and building the BDDs, while the execution of the EM iterations took only about 6 seconds for DBPedia and about 2 seconds for EDU-UK. For computing ARs' confidence, for each AR (that corresponds to the subclass axiom $A \sqsubseteq B$) two SPARQL queries have been executed over the training KBs, one for finding all the individuals that belong to $A \sqcap B$ and one for those that belong to A. The confidence is then given by the ratio of the number of individuals in $A \sqcap B$ over those in A. GoldMiner needed 330 different SPARQL queries for EDU-UK and 2,243 for DBPedia and took about 10,500 seconds for the first dataset and more than 578,000 seconds for the latter.

In the testing phase, we computed the probability of the queries using BUNDLE, according to the theory learned by EDGE and to the theory composed of the ARs with the confidence as probability. For a negative example of the form $a : C$, we computed the probability p of $a : C$ and we assigned probability $1 - p$ to the example.

We drew the Precision-Recall (PR) and the Receiver Operating Characteristics (ROC) curves and computed the Area Under the Curve (AUCPR and AUCROC) following the methods of [32, 41]. Table 26.1 shows the AUCPR, the AUCROC together with the standard deviation, the execution times averaged over the five folds and the p-value of a paired two-tailed t-test at the 5% significance level of the difference in AUCROC and AUCPR. The times are referred to the learning time for EDGE and to the SPARQL queries execution time for ARs.

Note that the elapsed time for EDGE depends on the number of executed

queries and the number of different explanations involved in each query, while the elapsed time for ARs depends on the number of classes in the KB. EDGE achieves greater areas in a time that is of the same or lower order of magnitude with respect to ARs. For both areas and KBs, the differences are statistically significant at the 5% level.

26.2 LEAP & EDGE: a Comparison Between Different Learning Problems

LEAP has been evaluated on three KBs:

- Carcinogenesis[3] [137] 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) [116] defining core concepts of requirements engineering and the way they are interrelated. It contains 107 individuals and 926 axioms.
- The Moral[4] 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. 8) in order

[3] `http://dl-learner.org/wiki/Carcinogenesis`
[4] `https://archive.ics.uci.edu/ml/datasets/Moral+Reasoner`

Table 26.2: Results of the experiments in terms of AUCPR and AUCROC averaged over the folds. 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

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 26.2 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, 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.

26.3 EDGE^{MR}: Parallelization Speedup

In order to evaluate the performances of EDGE^{MR}, we selected four datasets: three datasets from previous sections, i.e., Carcinogenesis, DBPedia and EDU-UK, and Mutagenesis[5] [138], containing 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.

For the generation of positive and negative examples, we followed the same approach explained in the previous sections. Once generated, positive and 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 26.3.

Table 26.3: Characteristics of the datasets used for evaluation.

	Carcinogenesis	DBPedia	EDU-UK	Mutagenesis
# of prob. axioms	186	1379	217	92
% of prob. axioms (# of axioms)	0.2% (74409)	25% (5380)	3% (5467)	0.1% (48354)
# of pos. examples	103	181	961	500
# of neg. examples	154	174	966	500
Fold size (MiB)	18.64	0.98	1.03	6.01

We performed the experiments with 1, 3, 5, 9 and 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 parameters found with 1 node are the same as those found with n nodes. Table 26.4 shows the running time in seconds for parameter learning on the

[5]http://www.doc.ic.ac.uk/ shm/mutagenesis.html

four datasets with the different configurations. Figure 26.1 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 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 achieves generally better performances than single-step scheduling.

Table 26.4: 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
EDU-UK	6924.2	3878	2157	1086	623	3612	2290	1332	749
Mutagenesis	1439.4	636	400	223	130	578	359	230	125

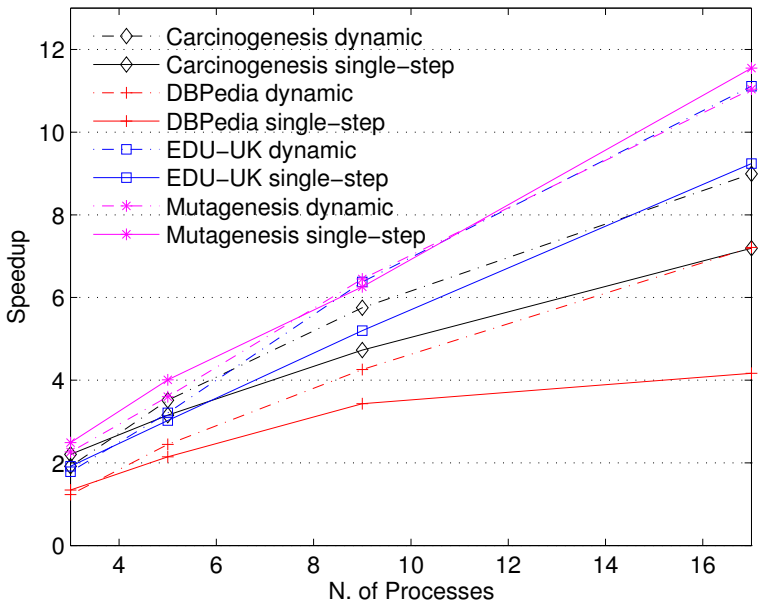
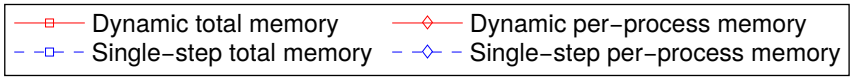


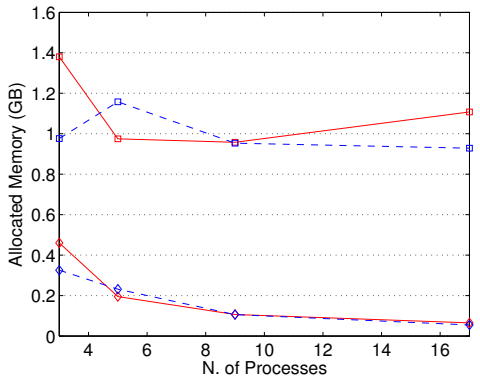
Figure 26.1: Speedup of EDGE$^{\text{MR}}$ relative to EDGE with single-step and dynamic schedulings.

26.4 EDGE^{MR}: Memory Consumption

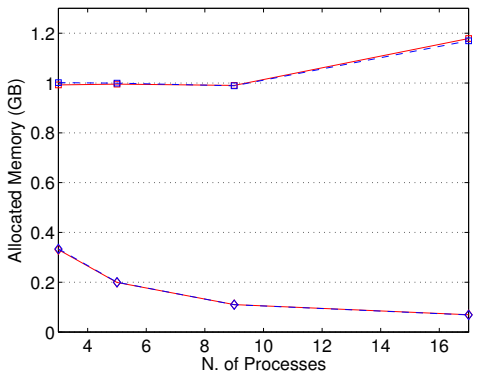
We also tested memory consumption on Carcinogenesis, DBPedia, EDU-UK and Mutagenesis. The configuration is as in Section 26.3. The results, shown in Fig. 26.2, show that the allocated memory per node is almost always inversely proportional to the number of nodes. There is no difference between Single-step and Dynamic scheduling in terms of used memory.



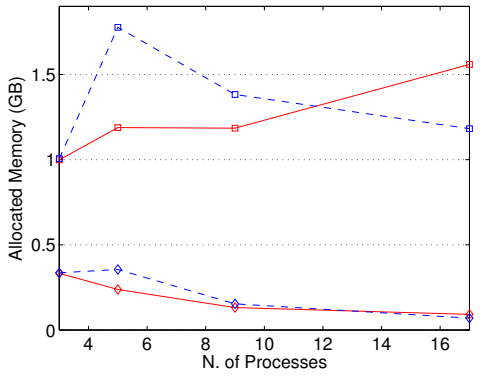
(a) Legend for memory consumption graphs



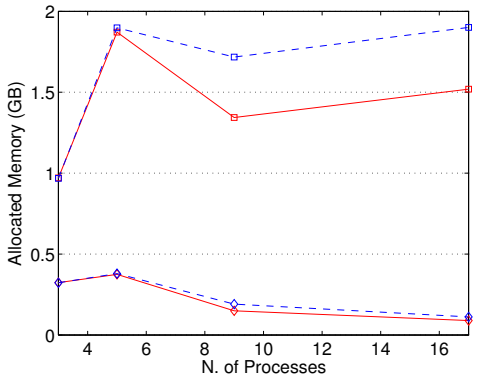
(b) Carcinogenesis



(c) DBPedia



(d) EDU-UK



(e) Mutagenesis

Figure 26.2: Memory consumption of EDGE^{MR} for different datasets.

26.5 LEAP^{MR}: Parallelization Speedup

In order to test how much the exploitation of EDGE^{MR} can improve the performances of LEAP^{MR}, we did a preliminary test where we considered only the Moral KB. We recall that it contains 202 individuals and 4710 axioms (22 axioms are probabilistic).

We allotted 1, 3, 5, 9 and 17 nodes, where the execution with 1 node corresponds to the 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 were generated by using CELOE and a maximum of 3 explanations per query was set for EDGE^{MR}. Figure 26.3 shows the speedup obtained as a function of the number of machines (nodes). As in Section 26.3, the speedup is sublinear but still quite good.

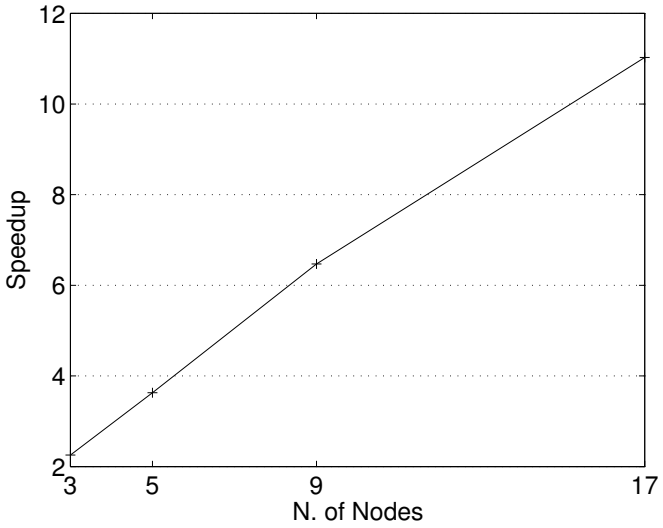


Figure 26.3: Speedup of LEAP^{MR} relative to LEAP for Moral KB.

26.6 Discussion

EDGE has been compared with ARs. In particular, each AR was regarded as a subclass axiom where the confidence was its probability. The results show that EDGE achieves larger areas both under the PR and the ROC curves with respect to an algorithm based on ARs in a comparable or smaller time. These

good results are partly due to the use of the BDDs built during inference, which allow to efficiently compute the expectations for hidden variables. In fact, these variables are not taken into account by ARs. Moreover, we found that the learning time of the two systems are comparable but behave differently, as shown in Table 26.1. This is due to the different operations executed: EDGE builds BDDs from explanations, hence the time depends on the KB, while ARs has to run a number of SPARQL queries depending on the number of concepts, roles and individuals in the KB. Anyway, we found that EDGE is a viable alternative to ARs.

In order to understand how structure learning algorithms can improve the quality of KBs, we did a simple but effective test by comparing three different KBs with their new versions returned by LEAP. The comparison was done by means of PR and ROC curves, thus we first made probabilistic the KBs through EDGE. The areas under both curves computed for LEAP were always greater than those of EDGE. LEAP increased the areas up to $\sim$840%.

Encouraged by these results, we started to study improvements for EDGE and LEAP, this led to the implementation of EDGE$^{\mathrm{MR}}$ and LEAP$^{\mathrm{MR}}$. Tests made on them show that the distribution of the computational load is effective, since the speedup is always greater than 1. Finally, EDGE$^{\mathrm{MR}}$ is equipped with two different scheduling techniques of which dynamic scheduling usually performs better. Memory consumption is comparable in the two scheduling techniques and in many cases they need almost the same amount of memory. In our tests, the difference on the values of the consumed memory always remained under 67%.

Part VI

Summary and Future Work

Chapter 27

Conclusion

Recently, the Semantic Web has become a reality and many domains have been modeled. The diffusion of the Semantic Web showed that coping with uncertain information is of foremost importance. Thus Probabilistic Description Logics have received an increased attention. Various semantics have been proposed but there is a lack of systems able to manage them.

The aim of this thesis was to provide a complete framework for managing uncertainty in the Semantic Web, by giving the definition of a probabilistic semantics and applications to work with it.

The proposed semantics, called DISPONTE, can be applied to every DL language. It minimally extends the language and allows the representation of *epistemic* probability, i.e., degrees of belief. DISPONTE is based on the distribution semantics, a well-known semantics which underlies many Logic Programming languages, and applies it to DLs. The distribution semantics defines a probability distribution on *possible worlds*, the probability of a query can be computed by finding a set of explanations, also called MinAs, which are then made mutually incompatible by means of *knowledge compilation* and in particular by building Binary Decision Diagrams (BDDs). A BDD permits the computation of the probability of the query in a time linear in its size. To be useful, a semantics also needs reasoning and learning systems. We thus propose inference and learning approaches.

Reasoning The system BUNDLE computes the probability of queries from a probabilistic KB following DISPONTE. It first finds the set of explanations for the given query and then builds the corresponding BDD.

The same approach is exploited in TRILL. Both implement a tableau algorithm, but they differ in the programming paradigm used. While BUNDLE is implemented in Java and has to handle *non-determinism* through an ad-hoc algorithm, TRILL is completely written in Prolog, thus the management of non-determinism is demandated to the Prolog's backtracking facilities.

Preliminary tests which compared BUNDLE and TRILL showed that a Prolog implementation of a DL reasoner is feasible and may be a new promising area to explore. Therefore, we continued on this line by implementing a third reasoning system, TRILL^P. It is written in Prolog as TRILL but, differently from the other reasoners, TRILL^P directly builds a monotone Boolean formula, called *pinpointing formula*, which compactly encodes the set of MinAs. Then, knowledge compilation by means of BDDs is applied to the pinpointing formula to compute the probability of queries.

The complexity of the three algorithms in the worst case is high since explanations may grow exponentially and the computation of the probability through Binary Decision Diagrams has a #P-complexity in the number of explanations. Nevertheless, experiments showed that domains of significant size can be managed.

Learning We have presented the EDGE system, which learns probability parameters in DLs exploiting an Expectation Maximization algorithm. It calls BUNDLE to build a BDD for each example, from which the values of expectations are directly computed. Experimental results over several real world datasets showed superior performances in term of quality of the results than an approach using Association Rules. Starting from these results, we developed LEAP, a supervised learning system able to learn both the structure and the parameters of a DL KB. LEAP exploits CELOE for creating descriptions of the target concept and EDGE to both test the quality of the descriptions and learn/tune the parameters of the resulting KB. We experimented whether a structure learning approach can improve the KBs. The tests showed that LEAP can achieve better results than simply tuning the parameters of an existing KB.

The diffusion of *Big Data* and the increased importance of *Linked Open Data* imply that standard serial algorithms cannot manage such huge amounts of data. Parallelization and distribution techniques must be used to cope with these issues. EDGE$^{\text{MR}}$ takes inspiration from MapReduce to distribute the computational load to different workers. In particular, building the examples' BDDs and the expectation step are split on the workers which run in parallel. The communication is performed using the Message Passing Interface standard. MapReduce frameworks such as Hadoop were not used since they commonly require purely functional operations, while we believe that keeping in memory part of the information is beneficial. The system LEAP$^{\text{MR}}$ exploits EDGE$^{\text{MR}}$ to speed up the learning time.

Tests made on the two distributed systems show that the parallelization is effective.

Overall, we believe that the interplay between the Semantic Web and Machine Learning opens extremely promising direction for the evolution of both.

Chapter 28

Future Work

In the future, we paln to optimize, improve and add functionalities to the inference systems, in particular

- develop a BUNDLE plug-in for the KB editor Protégé and a Web interface for BUNDLE similar to TRILL on SWISH;
- optimize TRILL and TRILL^P, for example by modifying the representation of the tableau by making use of dictionaries instead of red-black trees and replace the list representing the ABox with a structured representation such as a graph.

All the optimization will be evaluated to test the improvements produced. In particular, TRILL and TRILL^P might highly benefit from optimizations. Moreover, we plan to also explore different approaches, such as abduction. In preliminary works [49, 48], we considered an Abductive Logic Programming framework named $\mathcal{S}$CIFF, derived from the IFF abductive framework [47], able to deal with existentially (and universally) quantified variables in rule heads, and Constraint Logic Programming constraints. Forward and backward reasoning is naturally supported in $\mathcal{S}$CIFF. We showed that $\mathcal{S}$CIFF smoothly supports the integration of rules, expressed in a Logic Programming language, with Datalog$^{\pm}$ ontologies, mapped into $\mathcal{S}$CIFF (forward) integrity constraints. Datalog$^{\pm}$ can be used for representing lightweight ontologies, and is able to express the DL-Lite family of ontology languages, with tractable query answering under certain language restrictions. Thus, a DL-Lite KB can be translated

into a $\mathcal{S}$CIFF program through Datalog$^{\pm}$ and abductive reasoning can be applied on it. Preliminary tests showed this is a viable approach which however must be further investigated.

As regard learning, we would like to:

- improve even more the scalability of our algorithms, in order to handle larger datasets;
- integrate our learning systems in well-known state-of-art learning frameworks such as DL-Learner;
- allow our systems to automatically retrieve information on-line via public end-points, such as SPARQL servers;
- better evaluate our algorithms, in particular with other real world case studies.

Finally, we are also working in the area of Probabilistic Logic Programming (PLP) because we are convinced that the two fields are strictly intertwined, since the advances achieved in each of them can improve the other. Therefore, we are studying also PLP learning algorithms, such as SLIPCOVER and SEMPRE, mentioned in this thesis, or PASCAL (for "ProbAbiliStic inductive ConstrAint Logic") [121], where the distribution semantics was applied to Inductive Constraint Logic in order to learn (probabilistic) $\mathcal{S}$CIFF constraints. A probabilistic constraint logic models assign a probability of being positive to interpretations. This probability can be computed in a time that is logarithmic in the number of ground instantiations of violated constraints. PASCAL can learn both structure and parameters of these models. Some techniques implemented in these algorithms may be applied also to the systems presented in this thesis.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[2] A. Agresti and B. A Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[4] F. Baader and R. Peñaloza. Automata-based axiom pinpointing. *Journal of Automated Reasoning*, 45(2):91–129, 2010.

[5] F. Baader and R. Peñaloza. Axiom pinpointing in general tableaux. *Journal of Logic and Computation*, 20(1):5–34, 2010.

[6] F. Baader, R. Peñaloza, and B. Suntisrivaraporn. Pinpointing in the description logic EL^+. In *KI 2007: Advances in Artificial Intelligence, Proceedings of 30th Annual German Conference on AI*, volume 4667 of *LNCS*, pages 52–67. Springer, 2007.

[7] F. Bacchus. *Representing and reasoning with probabilistic knowledge - a logical approach to probabilities*. MIT Press, Cambridge, MA, USA, 1990.

[8] F. Bacchus. Using first-order probability logic for the construction of bayesian networks. In *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence*, pages 219–226. Morgan Kaufmann, 1993.

[9] C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1):57–144, 2009.

[10] B. Beckert and J. Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[11] E. Bellodi, E. Lamma, F. Riguzzi, V. S. Costa, and R. Zese. Lifted variable elimination for probabilistic logic programming. *Theory and Practice of Logic Programming*, Special Issue on the International Conference on Logic Programming(CoRR abs/1405.3218), 2014.

[12] E. Bellodi and F. Riguzzi. Learning the structure of probabilistic logic programs. In *22nd International Conference on Inductive Logic Programming*, volume 7207 of *LNCS*, pages 61–75. Springer Berlin Heidelberg, 2012.

[13] E. Bellodi and F. Riguzzi. Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2):343–363, 2013.

[14] E. Bellodi and F. Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(2):169–212, 2015.

[15] F. Bobillo and U. Straccia. fuzzyDL: An expressive fuzzy description logic reasoner. In *FUZZ-IEEE 2008, IEEE International Conference on Fuzzy Systems*, pages 923–930. IEEE, 2008.

[16] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[17] S. Bragaglia and F. Riguzzi. Approximate inference for logic programs with annotated disjunctions. In *Inductive Logic Programming - 20th*

International Conference, volume 6489 of *LNCS*, pages 30–37. Springer, 2010.

[18] M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. ProbLog technology for inference in a probabilistic first order logic. In *19th European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press, 2010.

[19] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 77–86. ACM, 2009.

[20] 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*, pages 95–130. Springer, 2009.

[21] R. N. Carvalho, K. B. Laskey, and P. C. G. da Costa. PR-OWL 2.0 - Bridging the gap to OWL semantics. In *Proceedings of the 6th International Workshop on Uncertainty Reasoning for the Semantic Web*, volume 654 of *CEUR Workshop Proceedings*, pages 73–84. CEUR-WS.org, 2010.

[22] İ. İ. Ceylan and R. Peñaloza. Bayesian description logics. In *Informal Proceedings of the 27th International Workshop on Description Logics*, volume 1193 of *CEUR Workshop Proceedings*, pages 447–458. CEUR-WS.org, 2014.

[23] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[24] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1-2):4–20, 2006.

[25] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[26] A. Choi and A. Darwiche. Relax, compensate and then recover. In *New Frontiers in Artificial Intelligence*, pages 167–180. Springer, 2011.

[27] M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.

[28] P. C. G. da Costa, K. B. Laskey, and K. J. Laskey. PR-OWL: A Bayesian ontology language for the Semantic Web. In *Uncertainty Reasoning for the Semantic Web I*, volume 5327 of *LNCS*, pages 88–107, Berlin, 2008. Springer.

[29] C. d'Amato, N. Fanizzi, and T. Lukasiewicz. Tractable reasoning with Bayesian Description Logics. In *2nd International Conference Scalable Uncertainty Management*, volume 5291 of *LNCS*, pages 146–159, Berlin, 2008. Springer.

[30] E. Dantsin. Probabilistic logic programs and their semantics. In *2nd Russian Conference on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.

[31] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intellingence Research*, 17:229–264, 2002.

[32] J. Davis and M. Goadrich. The relationship between Precision-Recall and ROC curves. In *23rd International Conference of Machine Learning*, pages 233–240. ACM, 2006.

[33] L. De Raedt, K. Kersting, A. Kimmig, K. Revoredo, and H. Toivonen. Compressing probabilistic Prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.

[34] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *20th International Joint Conference on Artificial Intelligence*, volume 7, pages 2462–2467. AAAI Press, 2007.

[35] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[36] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B*, 39(1):1–38, 1977.

[37] N. Di Mauro, E. Bellodi, and F. Riguzzi. Bandit-based monte-carlo structure learning of probabilistic logic programs. *Machine Learning*, 100(1):127–156, 2015.

[38] Z. Ding and Y. Peng. A probabilistic extension to ontology language OWL. In IEEE, editor, *Proceedings of the 37th Annual Hawaii International Conference On System Sciences*, pages 1–10. IEEE Computer Society Press, 2004.

[39] N. Eén and N. Sörensson. An extensible sat-solver. In *6th International Conference of Theory and Applications of Satisfiability Testing, Selected Revised Papers*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[40] I. Faizi. A Description Logic Prover in Prolog, Bachelor's thesis, Informatics Mathematical Modelling, Technical University of Denmark, 2011.

[41] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[42] D. Fierens, G. Van den Broeck, J. Renkens, D. Sht. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[43] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *27th Conference on Uncertainty in Artificial Intelligence*, pages 211–220. Morgan Kaufmann, 2011.

[44] D. Fleischhacker and J. Völker. Inductive learning of disjointness axioms. In *Confederated International Conferences On the Move to Meaningful Internet Systems*, volume 7045 of *LNCS*, pages 680–697. Springer, 2011.

[45] N. Friedman. The Bayesian Structural EM algorithm. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 129–138. Morgan Kaufmann, 1998.

[46] N. Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51(2):95–110, 2000.

[47] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[48] 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*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[49] 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*, volume 1459 of *CEUR Workshop Proceedings*, pages 128–143. CEUR-WS.org, 2015.

[50] L. Getoor, N.Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In *Proceedings of the 18th International Conference on Machine Learning*, pages 170–177. Morgan Kaufmann, 2001.

[51] 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*, volume 2424 of *LNCS*, pages 86–97. Springer, 2002.

[52] T. Gomes and V. S. Costa. Evaluating inference algorithms for the prolog factor language. In *22nd International Conference on Inductive Logic Progamming*, volume 7842 of *LNCS*, pages 74–85. Springer, 2012.

[53] G. Gottlob, T. Lukasiewicz, and G. I. Simari. Conjunctive query answering in probabilistic Datalog+/- ontologies. In *Proceedings of the 5th International Conference on Web Reasoning and Rule Systems*, volume 6902 of *LNCS*, pages 77–92. Springer, 2011.

[54] O. Grumberg, S. Livne, and S. Markovitch. Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116, 2003.

[55] 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 Knowledge Discovery in Databases*, volume 5211 of *LNCS*, pages 473–488. Springer, 2008.

[56] C. Halaschek-Wiener, A. Kalyanpur, and B. Parsia. Extending tableau tracing for ABox updates. Technical report, University of Maryland, 2006.

[57] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990.

[58] J. Heinsohn. Probabilistic Description Logics. In *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence*, pages 311–318. Morgan Kaufmann Publishers Inc., 1994.

[59] T. Herchenröder. Lightweight semantic web oriented reasoning in Prolog: Tableaux inference for description logics. Master's thesis, School of Informatics, University of Edinburgh, 2006.

[60] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. CRCPress, 2009.

[61] B. Hollunder. An alternative proof method for possibilistic logic and its application to terminological logics. *International Journal of Approximate Reasoning*, 12(2):85–109, 1995.

[62] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible SROIQ. In *Proceedings of the 10th International Conference on Principles of*

Knowledge Representation and Reasoning, pages 57–67. AAAI Press, 2006.

[63] U. Hustadt, B. Motik, and U. Sattler. Deciding expressive description logics in the framework of resolution. *Information and Computation*, 206(5):579–601, 2008.

[64] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the em algorithm by bdds. Technical Report TR08-0004, Dep. of Computer Science, Tokyo Institute of Technology, 2008.

[65] M. Jaeger. Probabilistic reasoning in terminological logics. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 305–316. Morgan Kaufmann Publishers Inc., 1994.

[66] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[67] J. C. Jung and C. Lutz. Ontology-based access to probabilistic data with OWL QL. In *Proceedings of the 11th International Semantic Web Conference, Part I*, volume 7649 of *LNCS*, pages 182–197. Springer, 2012.

[68] A. Kalyanpur. *Debugging and Repair of OWL Ontologies*. PhD thesis, The Graduate School of the University of Maryland, 2006.

[69] A. Kalyanpur, B. Parsia, B. Cuenca-Grau, and E. Sirin. Tableaux tracing in SHOIN. Technical Report 2005-66, University of Maryland, 2005.

[70] A. Kalyanpur, B. Parsia, M. Horridge, and E. Sirin. Finding all justifications of OWL DL entailments. In *6th International Semantic Web Conference and 2nd Asian Semantic Web Conference*, volume 4825 of *LNCS*, pages 267–280. Springer, 2007.

[71] A. Kalyanpur, B. Parsia, E. Sirin, and J. A. Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics*, 3(4):268–293, 2005.

[72] K. Kersting and L. De Raedt. Towards combining inductive logic programming with bayesian networks. In *11th International Conference on Inductive Logic Programming*, volume 2157 of *LNCS*, pages 118–131. Springer, 2001.

[73] A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[74] P. Klinov. Pronto: A non-monotonic probabilistic Description Logic reasoner. In *Proceedings of the 5th European Semantic Web Conference, The Semantic Web: Research and Applications*, volume 5021 of *LNCS*, pages 822–826. Springer, 2008.

[75] P. Klinov and B. Parsia. Optimization and evaluation of reasoning in probabilistic Description Logic: Towards a systematic approach. In *Proceedings of the 7th International Semantic Web Conference*, volume 5318 of *LNCS*, pages 213–228. Springer, 2008.

[76] P. Klinov and B. Parsia. A hybrid method for probabilistic satisfiability. In *Proceedings of the 23rd International Conference on Automated Deduction*, volume 6803 of *LNCS*, pages 354–368. Springer, 2011.

[77] D. Koller, A. Y. Levy, and A. Pfeffer. P-CLASSIC: A tractable probabilistic Description Logic. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference*, pages 390–397. AAAI Press / The MIT Press, 1997.

[78] D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *International Joint Conference on Artifical Intelligence*, volume 2, pages 1316–1321. Morgan Kaufmann, 1997.

[79] R. Kowalski. Predicate logic as programming language. In *International Federation for Information Processing Congress*, volume 74, pages 569–544, 1974.

[80] T. Lager and J. Wielemaker. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming*, 14(4-5):539–552, 2014.

[81] 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*, pages 346–353. AUAI Press, 2005.

[82] D. J. Lehmann. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence*, 15(1):61–82, 1995.

[83] J. Lehmann, S. Auer, L. Bühmann, and S. Tramp. Class expression learning for ontology engineering. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(1):71–81, 2011.

[84] J. Lehmann and P. Hitzler. Concept learning in description logics using refinement operators. *Machine Learning*, 78:203–250, 2010.

[85] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[86] G. Lukácsy and P. Szeredi. Efficient description logic reasoning in prolog: The DLog system. *Theory and Practice of Logic Programming*, 9(3):343–414, 2009.

[87] T. Lukasiewicz. Probabilistic default reasoning with conditional constraints. *Annals of Mathematics and Artificial Intelligence*, 34(1-3):35–88, 2002.

[88] T. Lukasiewicz. Expressive probabilistic description logics. *Artificial Intelligence*, 172(6-7):852–883, 2008.

[89] T. Lukasiewicz and U. Straccia. An overview of uncertainty and vagueness in description logics for the semantic web. Technical report, INFSYS RR-1843-06-07, Institut für Informationssysteme, Technische Universität Wien, 2006.

[90] J. E. Ochoa Luna and F. Gagliardi Cozman. An algorithm for learning with probabilistic description logics. In *Proceedings of the 5th International Workshop on Uncertainty Reasoning for the Semantic Web and 8th International Semantic Web Conference*, volume 527 of *CEUR Workshop Proceedings*, pages 63–74. CEUR-WS.org, 2009.

[91] J. E. Ochoa Luna, K. Revoredo, and F. Gagliardi Cozman. Learning probabilistic Description Logics: A framework and algorithms. In *Proceedings of the 10th Mexican International Conference on Artificial Intelligence, Advances in Artificial Intelligence, Part I*, volume 7094 of *LNCS*, pages 28–39. Springer, 2011.

[92] C. Lutz and L. Schröder. Probabilistic Description Logics for subjective uncertainty. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning*, pages 393–403. AAAI Press, 2010.

[93] W. Meert, J. Struyf, and H. Blockeel. Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, 89(1):131–160, 2008.

[94] A. Meissner. An automated deduction system for description logic with alcn language. *Studia z Automatyki i Informatyki*, 28-29:91–110, 2004.

[95] 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*, pages 378–383. ACM, 2012.

[96] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[97] R. T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[98] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171(1-2):147–177, 1997.

[99] M. Niepert, J. Noessner, and H. Stuckenschmidt. Log-linear description logics. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2153–2158. IJCAI/AAAI, 2011.

[100] N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.

[101] F. Patel-Schneider, P, I. Horrocks, and S. Bechhofer. Tutorial on OWL, 2003.

[102] R. Peñaloza and B. Sertkaya. Axiom pinpointing is hard. In *Proceedings of the 22nd International Workshop on Description Logics*, volume 477 of *CEUR Workshop Proceedings*, pages 1–12. CEUR-WS.org, 2009.

[103] R. Peñaloza and B. Sertkaya. Complexity of axiom pinpointing in the DL-Lite family. In *Proceedings of the 23rd International Workshop on Description Logics*, volume 573 of *CEUR Workshop Proceedings*, pages 1–12. CEUR-WS.org, 2010.

[104] R. Peñaloza 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*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 29–34. IOS Press, 2010.

[105] D. Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3):377–400, 1993.

[106] D. Poole. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[107] D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[108] D. Poole. Abducing through negation as failure: stable models within the Independent Choice Logic. *Journal of Logic Programming*, 44(1-3):5–35, 2000.

[109] D. Poole. First-order probabilistic inference. In *18th International Joint Conference on Artificial Intelligence*, pages 985–991. Morgan Kaufmann, 2003.

[110] G. Qi, Q. Ji, J. Z. Pan, and J. Du. Possdl - A possibilistic DL reasoner for uncertainty reasoning and inconsistency handling. In *Proceedings of the 7th Extended Semantic Web Conference, The Semantic Web: Research and Applications, Part II*, volume 6089 of *LNCS*, pages 416–420. Springer, 2010.

[111] A. Rauzy, E. Châtelet, Y. Dutuit, and C. Bérenguer. A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety*, 79(1):33–42, 2003.

[112] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[113] J. Renkens, A. Kimmig, G. Van den Broeck, and L. De Raedt. Explanation-based approximate weighted model counting for probabilistic logics. In *28th AAAI Conference on Artificial Intelligence*, pages 2490–2496. AAAI Press, 2014.

[114] K. Revoredo, J. Eduardo Ochoa Luna, and F. Gagliardi Cozman. Learning terminologies in probabilistic description logics. In *Proceedings in the 20th Brazilian Symposium on Artificial Intelligence, Advances in Artificial Intelligence*, volume 6404 of *LNCS*, pages 41–50. Springer, 2010.

[115] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, and N. Leone. OntoDLV: An ASP-based system for enterprise ontologies. *Journal of Logic and Computation*, 19(4):643–670, 2009.

[116] T. Riechert, K. Lauenroth, J. Lehmann, and S. Auer. Towards semantic based requirements engineering. In *Proceedings of the 7th International Conference on Knowledge Management*, pages 144–151, 2007.

[117] F. Riguzzi. A top down interpreter for LPAD and CP-logic. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNAI*, pages 109–120. Springer, 2007.

[118] F. Riguzzi. Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009.

[119] F. Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013.

[120] F. Riguzzi. Speeding up inference for probabilistic logic programs. *The Computer Journal*, 57(3):347–363, 2014.

[121] F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. Probabilistic inductive constraint logic. In *25th International Conference on Inductive Logic Programming*, 2015.

[122] F. Riguzzi and N. Di Mauro. Applying the information bottleneck to statistical relational learning. *Machine Learning*, 86(1):89–114, 2012.

[123] F. Riguzzi and T. Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *26th International Conference on Logic Programming*, volume 7 of *LIPIcs*, pages 162–171, 2010.

[124] F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4–5):433–449, 2011.

[125] F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(Special Issue 02 - 25th Annual GULP Conference):279–302, 2013.

[126] V. S. Costa, D. Page, M. Qazi, and J. Cussens. CLP(BN): Constraint logic programming for probabilistic knowledge. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, pages 517–524. Morgan Kaufmann, 2003.

[127] V. S. Costa, R. Rocha, and L. Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[128] T. Sang, P. Beame, and H. A. Kautz. Performing bayesian inference by weighted model counting. In *20th National Conference on Artificial Intelligence*, pages 475–482, 2005.

[129] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming*, pages 715–729. MIT Press, 1995.

[130] T. Sato. Generative modeling by PRISM. In *Proceedings of the 25th International Conference on Logic Programming*, volume 5649 of *LNCS*, pages 24–35. Springer, 2009.

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

[132] U. Sattler, D. Calvanese, and R. Molitor. Relationships with other formalisms. In *Description Logic Handbook*, pages 137–177, 2003.

[133] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 355–362. Morgan Kaufmann Publishers Inc., 2003.

[134] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[135] G. Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

[136] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[137] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In *7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNCS*, pages 273–287. Springer Berlin Heidelberg, 1997.

[138] A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

[139] U. Straccia. A fuzzy description logic. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference*, pages 594–599. AAAI Press / The MIT Press, 1998.

[140] U. Straccia. Reasoning within fuzzy description logics. *Journal of Artificial Intelligence Research*, 14:137–166, 2001.

[141] S. Toda. On the computational power of PP and +P. In IEEE, editor, *30th Annual Symposium on Foundations of Computer Science*, pages 514–519. IEEE Computer Society Press, 1989.

[142] C. Tresp and R. Molitor. A description logic for vague knowledge. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 361–365, 1998.

[143] T. Tudorache, C. Nyulas, N. Fridman Noy, and M. A. Musen. Webprotégé: A collaborative ontology editor and knowledge acquisition tool for the web. *Semantic Web*, 4(1):89–99, 2013.

[144] URW3-XG. Uncertainty reasoning for the World Wide Web, final report, 2008. `http://www.w3.org/2005/Incubator/urw3/XGR-urw3`.

[145] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing (SICOMP)*, 8(3):410–421, 1979.

[146] G. Van den Broeck, W. Meert, and A. Darwiche. Skolemization for weighted first-order model counting. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 2014.

[147] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[148] 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*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[149] J. Vennekens, M. Denecker, and M.Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.

[150] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic Programs With Annotated Disjunctions. In *20th International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer Berlin Heidelberg, 2004.

[151] J. Völker and M. Niepert. Statistical schema induction. In *8th Extended Semantic Web Conference*, volume 6643 of *LNCS*, pages 124–138. Springer, 2011.

[152] M. P. Wellman, J. S. Breese, and R. P. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(01):35–53, 1992.

[153] J. Wielemaker, Z. Huang, and L. van der Meij. SWI-Prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.

[154] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[155] P. M. Yelland. An alternative combination of bayesian networks and description logics. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 225–234. Morgan Kaufmann, 2000.