Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN
SCIENZE DELL'INGEGNERIA

CICLO XXXIV

COORDINATORE Prof. Trillo Stefano

Extensions and Applications of Probabilistic Logic Programming

Settore scientifico disciplinare ING-INF/05

Dottorando

Dott. Azzolini Damiano

Tutori

Prof. Riguzzi Fabrizio
Prof.ssa Lamma Evelina

Anni 2018/2021

Acknowledgements

First of all, I would like to thank my supervisors Fabrizio Riguzzi and Evelina Lamma that guided me through this journey and let me explore different research topics.

I thank my family, Sandra, Andrea, and Francesca, for the unconditional support in all my activities.

In the last two years of the PhD I worked from home so I need to thank my grandma Bruna for preparing me food every day, even if she does not like cooking. I also thank my dog Rocky since he actually became my colleague, always listening to my complaints.

A special thanks also goes to my girlfriend Federica that always brings shine in my days full of obscure papers and meaningless algorithms.

Damiano Azzolini

The only man who never makes a mistake
is the man who never does anything.

Contents

I Introduction **1**

1 Motivation **3**

2 Goal of the Thesis **5**

3 Structure of the Thesis **7**

3.1 How to Read this Thesis 10

II Background **11**

4 Probability Theory and Set Theory **13**

4.1 Set Theory 13

4.2 Probability Theory 15

4.2.1 Random Variables 17

4.3 Ordinal Numbers, Mappings, and Fixpoints 20

4.3.1 Ordinal Numbers 20

4.3.2 Mappings and Fixpoints 21

5 Logic and Logic Programming **23**

5.1 Propositional and First Order Logic 23

5.1.1 Propositional Logic 23

5.1.2 First Order Logic 25

5.2 Logic Programming 30

5.2.1 Semantics for Programs with Negation 34

5.3 Abduction and Abductive Logic Programming 38

6 Syntax and Semantics for Probabilistic Logic Programs 41
6.1 ProbLog and LPADs 41
6.2 Distribution Semantics 44
6.2.1 Semantics for Programs with Function Symbols 46
6.3 Conclusions 51

7 Inference 53
7.1 Exact Inference 53
7.1.1 Decision Diagrams 56
7.1.2 Systems to Perform Exact Probabilistic Logical Inference 57
7.2 Approximate Inference 59
7.2.1 Markov Chain Monte Carlo 61
7.3 Conclusions 76

III Extensions of Probabilistic Logic Programming 79

8 Hybrid Programs 81
8.1 Hybrid Probabilistic Logic Programs 81
8.1.1 Probabilistic Constraint Logic Programming 85
8.2 Semantics for Hybrid Programs with Function Symbols 93
8.2.1 A Concrete Syntax 108
8.2.2 Syntactic Requirements 113
8.3 Conclusions 117

9 Extending Probabilistic Logic Programming with Abduction 119
9.1 Probabilistic Abductive Logic Programs 119
9.1.1 Examples 123
9.1.2 Algorithm 129
9.1.3 Experiments 135
9.2 Related Work 139
9.2.1 Relation with MAP, MPE, and Viterbi 143
9.3 Conclusions 146

10 Integrating Constraints and Probability 147
10.1 Probabilistic Optimizable Logic Programs 147

10.1.1 Experiments 155
10.2 Probabilistic Reducible Logic Programs 158
10.2.1 Experiments 162
10.3 Related Work 166
10.4 Conclusions 168

IV Applications of Probabilistic Logic Programming 171

11 Blockchain 173
11.1 Structure 173
11.1.1 Bitcoin 174
11.1.2 Smart Contracts 178
11.1.3 Lightning Network 179

12 Analysis of Blockchain-related Scenarios 181
12.1 Smart Contract Analysis 181
12.1.1 Experiments 184
12.1.2 Conclusions 188
12.2 Hashing Power Centralization and Double Spending 189
12.2.1 Preventing the Formation of Large Pools 189
12.2.2 Double Spending 191
12.2.3 Conclusions 195
12.3 Transaction Fees 196
12.3.1 Analyzing Transaction Fees 197
12.3.2 Probability of a Profitable Fork 201
12.3.3 Related Work 208
12.3.4 Conclusions 208
12.4 Lightning Network Model 209
12.4.1 Deterministic Model 210
12.4.2 Probabilistic Model 215
12.5 Conclusions 220

V Conclusions and Outlooks **223**

13 Conclusions **225**

14 Future Work **227**

Bibliography **229**

List of Figures

3.1 Chapter dependency graph. 10

5.1 SLD tree for the query `reach(a,c)`. 32

7.1 Decision Diagrams. 58
7.2 Results for the arithm experiment. 66
7.3 Results for the HMM experiment. 67
7.4 Results for the LDA experiment. 68
7.5 Results for the LDA and university experiments. For both we fixed the number of samples to 10^4. 68
7.6 Results for the university experiment. 69
7.7 Results for the arithm experiment. 70
7.8 Results for the diabetes experiment. 71
7.9 Standard deviation for the arithm and diabetes experiments. . . 71
7.10 Results for the graph experiment. 72
7.11 Results for the HMM experiment. 73
7.12 Standard deviation for the graph and HMM experiments. 73
7.13 Results fot the LDA experiment. 74
7.14 Results for the nballs experiment. 74
7.15 Standard deviation for the LDA and nballs experiments. 75
7.16 Results for the prefix parser experiment. 75
7.17 Results for the stochastic logic program experiment. 76
7.18 Standard deviation for the prefix parser and stochastic logic program experiments. 76

9.1 Example program and its worlds. I and E indicate respectively whether the IC is included (I) or not (E) in each world. 121

9.2 Program, BDD, and worlds for Example 20 variant 1. Highlighted rows in the table represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{c,e}`, together with their probability. 124
9.3 BDD and worlds for Example 20 variant 2. Highlighted rows in the table represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{e}`, together with their probability. 124
9.4 BDDs for the example showing the conjunction of BDDs. 130
9.5 BDD and truth table for the example showing the conjunction of BDDs. Highlighted rows represent the combinations of arguments such that the expression `((a and d) or (b and c)) and (not(a and b))` (compactly referred as Expr in the table) is true. 131
9.6 Inference time as a function of the number of abducibles for the gh and gnb datasets, with and without integrity constraints. . . 139
9.7 Inference time as a function of the number of abducibles for the blood dataset, with and without integrity constraints. 139
9.8 Inference time as a function of the number of abducibles for the graph and complete graph datasets, with and without integrity constraints. 140

10.1 Program for the motivating example, together with the network graph. 150
10.2 BDD for the program shown in Figure 10.1, where a dashed line represents a 0-edge, a solid line the 1-edge, and a dotted line the 0-complemented edge. 152
10.3 Execution time for directed and undirected complete graphs. . . 165
10.4 Computed gaps of the approximate algorithm on both directed and undirected complete graphs. 165

12.1 Expected payout of a consecutive number of trials. λ represents the mean of the Poisson distribution. 188
12.2 Markov chain of the model. 190

12.3 Initial state of the double spending attack. Block $B1$ with transaction $T1$ is inserted in the chain after $B0$ while the attacker starts mining another block ($B2$) without $T1$ inside and with $B0$ as ancestor. 192
12.4 General case. The "honest" chain has built 3 confirmation blocks on top of $B1$ ($B3, B4, B5$) while only one block ($B6$) has been built on top of $B2$ by the attacker. In this figure, d represents the distance between the honest and the secret chain and is used to evaluate the advantage of the honest chain over the attacker. 192
12.5 Successful attack. The attacker has built a longer chain (marked in red). The attacker will now publish all blocks from $B2$ to $B9$ and so all blocks from $B1$ to $B5$ in the black chain will not be considered valid because they are part of a chain which is not the longest one. 192
12.6 Success probability of a double spending attack by considering Poisson and Pascal probability distributions for the number of blocks mined by the attacker. 195
12.7 Graphs relating the size of the block and the observed fee rate with the profit of a miner. Dashed lines represent the values computed with `mc_expectation/3` (without observations). . . . 198
12.8 The graph shows how transaction fees influence the probability of confirmation within 1 block. 201
12.9 Results for the threshold experiment. 205
12.10 Results for the value experiment. 206
12.11 Results for the optimal experiment. Each graph shows the difference between the expected value obtained from being honest and from forking the chain starting from a block with n times the average reward (σ) with different values of the fraction of the controlled mining power β. 207
12.12 Channel representation in the Lightning Network. Case (a) corresponds to (b) in practice since the distribution is unknown. . . 210
12.13 Nodes degree distribution and average maximum rebalancing amount. 212

12.14 Variation of the total network capacity by removing edges and nodes for the three LN states. 213
12.15 Number of paths and non redundant paths of length 2 and 3 between equal degree nodes. 214
12.17 Probability of a successful payment of varying size between random nodes. 218
12.18 Probability of a successful payment split in multiple parts between nodes of various degrees where intermediate nodes are always active. 219
12.19 Probability of a successful payment split in multiple parts between nodes of various degrees. 219
12.20 Probability of a successful payment between nodes of various degrees when intermediate nodes could be inactive. 220

List of Tables

6.1 Worlds for a restricted version of the ProbLog program of Example 1, where the probabilistic fact `loss_fact/1` and the two clauses `loss/0` and `draw/0` have been removed. Highlighted rows represent the worlds where the query `win` is true, together with their probability. 46

9.1 Worlds for Example 20 variant 3. Highlighted rows represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{e}`, together with their probability. I and E stand respectively for included and excluded. 125

9.2 Possible worlds for the LPAD of Example 24 (Variant 3) with the corresponding probability, computed as the product of the probabilities associated with the head atoms taking value true, reported in each row. Highlighted rows represent the worlds in which the query `eruption` is true. 128

9.3 Possible worlds for the LPAD of Example 24 with the corresponding probability computed as the product of the probabilities associated with the head atoms taking value true, reported in each row. Highlighted rows represent the worlds in which the query `eruption` is true. 129

9.4 Details of the datasets. 138

10.1 Results for the network experiments. C, M and S stand respectively for CCSAQ, MMA, and SLSQP algorithms. `|V|` is the number of vertices and `|E|` the number of edges respectively. . . 157

10.2 Results for the complete graphs experiments. 157

10.3 Execution time for the probabilistic graph dataset of [32]. 166

10.4 Results for the exact algorithm on graphs from [143]. 166

12.1 Details for the transfer experiment without bug. 186
12.2 Details for the transfer experiment with bug. 186
12.3 Details for the pyramid experiment. 187
12.4 Resource usage for the gambling experiment with $\lambda = 150$. 188
12.5 Datasets structure for the three LN states (February, March, and April). Capacities are expressed in satoshi. 211
12.6 Capacity of the edges (expressed in satoshi) and frequency for February (F), March (M), and April (A). 212
12.7 Information about channel capacities (in satoshi) and node degrees. 216

List of Algorithms

1 Function PROB: computation of the probability of a BDD. . . . 58
2 Function REJECTIONSAMPLING: rejection sampling algorithm. . 61
3 Function MH: Metropolis-Hastings MCMC algorithm. 63
4 Function GIBBS: Gibbs MCMC algorithm. 65
5 Function ABDUCTIVEEXPL: computation of the *minimal* sets that maximize the joint probability of the query and the ICs, and of the corresponding probability. 131
6 Function ABDINT: computation of the sets that maximize the joint probability of the query and the ICs, and of the corresponding probability, through BDD exploration. 132
7 Function OPTIMIZEPROB: optimization of probability of random variables. 152
8 Function PATHSPROB: computation of all the paths of a BDD and of their probability. 153
9 Function MINIMIZEREDUCIBLES: minimizing the number of reducible facts. 163

Part I

Introduction

Chapter 1

Motivation

We are all over-connected through computers and mobile devices. Every action we do using these technologies is stored, and the amount of collected data is enormous and not easily interpretable. Moreover, collected data are often somehow related, and information can be inferred from them. For example, if I travel from Monday to Friday every morning at the same hour from one city to another city, it is likely that I will work in this second city, or that, at least, I have some recurrent activities to do there.

First Order Logic is a powerful and consolidated formalism to represent relational data, since it allows to explicitly state what relations hold. By means of Logic Programming, it is possible to extract some information from this massive amount of data using an easily interpretable language. Moreover, the field of Inductive Logic Programming aims at learning programs that model a desired scenario given a training set of examples. One limitation of Logic Programming is that it does not handle uncertainty, an intrinsic characteristic of data, since they often come from different sources with different levels of reliability.

The field of Statistical Relational Artificial Intelligence (often abbreviated with StarAI), aims to combine the expressivity of logic with uncertainty. One of the possible formalisms is Probabilistic Logic Programming (PLP), where Logic Programs are extended to cope with probabilistic information. Most PLP languages follow a precise semantics, the Distribution Semantics (DS), proposed in 1995, that defines how to perform logical reasoning in the presence of uncertainty. The DS has been proposed only for discrete data but often

measurements require continuous variables, such as temperature. Moreover, we often want to express constraints among the possible relations and also to deal with missing data. For these reasons, several extensions have been proposed during the years, to cover a broad spectrum of possible application scenarios.

Chapter 2

Goal of the Thesis

The goal of this dissertation is to provide new inference algorithms for probabilistic logic programs, to extend the possible tasks that can be solved with PLP, and to apply PLP in the context of the blockchain.

Due to the amount of available data, usually exact inference is infeasible. To alleviate this, we propose and implement two Markov Chain Monte Carlo algorithms, namely Gibbs sampling and Metropolis Hastings sampling, to perform approximate inference on PLP, and test them on several datasets.

Real-world data require managing both continuous and discrete measurements, but the proposed semantics for *hybrid* (i.e., with both discrete and continuous random variables) probabilistic logic programs have some restrictions. To allow inference on a large class of hybrid probabilistic logic programs, we propose a new semantics, prove it well defined, and provide several syntactic requirements that must be respected to keep the semantics well-defined. Moreover, data are often incomplete, and reasoning in this case can be performed by abduction (hence, Abductive Logic Programming). We extend PLP with abduction, proposing probabilistic *abductive* logic programs to manage uncertain and incomplete data, and providing a practical inference algorithm whose effectiveness has been proved through a series of examples.

In many applications the goal is to learn the values of the parameters of a program describing a process, and sometimes even learn the program itself starting from data. These two tasks have been extensively studied in StarAI and they go under the name of parameter learning and structure learning respectively. However, the integration of these tasks, and probabilistic systems

in general, with constraints involving random variable values has not been yet extensively explored. Here, we propose two new classes of probabilistic logic programs: probabilistic *optimizable* logic program and probabilistic *reducible* logic programs. For the former, the goal is to set the probabilities of some facts such that an objective function is optimized and the imposed constraints are not violated (thus, parameter learning) while, for the latter, the goal is to remove facts from the program such that the imposed constraints are not violated (thus, a version of structure learning). For both, we provide two algorithms and test their performance on real-world datasets.

Among recent technology, the blockchain is certainly one that attracted a lot of interest due to several theoretical properties such as immutability and decentralization. The blockchain has numerous application scenarios, extensions, and critical issues. Here, we propose PLP models to study smart contracts, transaction fees, Lightning Network, and hashing power centralization.

Chapter 3

Structure of the Thesis

This thesis is divided into five parts: after the introduction (first part), the second part introduces the background concepts, the third part discusses the extensions of Probabilistic Logic Programming (PLP), the fourth part illustrates some possible applications of PLP, in particular in the context of blockchain, and the fifth part concludes this thesis with several possible directions for future work.

In details, the second part (background) is not novel, except for Section 7.2.1 that introduces an approximate inference technique based on Monte Carlo Markov Chain methods and described in the following publications:

- Damiano Azzolini, Fabrizio Riguzzi, Evelina Lamma, and Franco Masotti. A comparison of MCMC sampling for probabilistic logic programming. In Mario Alviano, Gianluigi Greco, and Francesco Scarcello, editors, Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy, pages 19–22, Lecture Notes in Computer Science, Heidelberg, Germany, 2019.

- Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. An analysis of Gibbs sampling for probabilistic logic programs. In Carmine Dodaro, George Aristidis Elder, Wolfgang Faber, Jorge Fandinno, Martin Gebser, Markus Hecher, Emily LeBlanc, Michael Morak, and Jessica Zangari, editors, Workshop on Probabilistic Logic Programming (PLP 2020), volume 2678 of CEUR Workshop Proceedings, pages 1–13, Aachen, Germany, 2020.

The third part, called "Extensions of Probabilistic Logic Programming", discusses, as the title says, some extensions of PLP. Chapter 8 introduces the semantics of hybrid programs (Section 8.2), i.e., programs that combine both discrete and continuous random variables, together with some syntactic requirements needed to keep it well-defined (Section 8.2.2), and is based on the following publications:

- Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. A semantics for hybrid probabilistic logic programs with function symbols. Artificial Intelligence, 294:103452, 2021.

- Damiano Azzolini and Fabrizio Riguzzi. Syntactic requirements for well-defined hybrid probabilistic logic programs. In Andrea Formisano, Yanhong Annie Liu, Bart Bogaerts, Alex Brik, Veronica Dahl, Carmine Dodaro, Paul Fodor, Gian Luca Pozzato, Joost Vennekens, and Neng-Fa Zhou, editors, Proceedings 37th International Conference on Logic Programming (Technical Communications), pages 14–26, Waterloo, Australia, 2021.

An extension of PLP that integrates it with abduction is described in Chapter 9, where the class of probabilistic abducible logic programs is introduced, based on this publication:

- Damiano Azzolini, Elena Bellodi, Stefano Ferilli, Fabrizio Riguzzi, and Riccardo Zese. Abduction with Probabilistic Logic Programming under the Distribution Semantics. International Journal of Approximate Reasoning, 142:41–63, 2022.

Two more extensions that integrate constraints and probability are discussed in Chapter 10: probabilistic optimizable logic programs (Section 10.1) and probabilistic reducible logic programs (Section 10.2). They are based on the following publications respectively:

- Damiano Azzolini and Fabrizio Riguzzi. Optimizing probabilities in probabilistic logic programs. Theory and Practice of Logic Programming, pages 543–556, 2021.

- Damiano Azzolini and Fabrizio Riguzzi. Reducing probabilistic logic programs. In Ahmet Soylu, Alireza Tamaddoni Nezhad, Nikolay Nikolov, Ioan Toma, Anna Fensel, and Joost Vennekens, editors, Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium at RuleML+RR 2021 co-located with 17th Reasoning Web Summer School (RW 2021) and 13th DecisionCAMP 2021 as part of Declarative AI 2021, CEUR Workshop Proceedings, pages 1–13, Aachen, Germany, 2021.

The fourth part is dedicated to the discussion of some applications of PLP in blockchain environments, and are based on these publications, listed in chronological order, starting from the oldest one:

- Damiano Azzolini, Fabrizio Riguzzi, Evelina Lamma, Elena Bellodi, and Riccardo Zese. Modeling bitcoin protocols with probabilistic logic programming. In Elena Bellodi and Tom Schrijvers, editors, Probabilistic Logic Programming (PLP 2018), volume 2219 of CEUR Workshop Proceedings, pages 49–61, Aachen, Germany, 2018.

- Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Studying transaction fees in the bitcoin blockchain with probabilistic logic programming. Information, 10(11):335.

- Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Analyzing transaction fees with probabilistic logic programming. In Witold Abramowicz and Rafael Corchuelo, editors, Business Information Systems Workshops BIS 2019, volume 373 of Lecture Notes in Business Information Processing, pages 243–254, Cham, 2019.

- Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Modeling smart contracts with probabilistic logic programming. In Witold Abramowicz and Gary Klein, editors, Business Information Systems Workshops, volume 394 of Lecture Notes in Business Information Processing, pages 86–98, Cham, 2020.

- Damiano Azzolini, Elena Bellodi, Alessandro Brancaleoni, Fabrizio Riguzzi, and Evelina Lamma. Modeling bitcoin lightning network by logic programming. In Francesco Ricca, Alessandra Russo, Sergio Greco, Nicola

Leone, Alexander Artikis, Gerhard Friedrich, Paul Fodor, Angelika Kimmig, Francesca Lisi, Marco Maratea, Alessandra Mileo, and Fabrizio Riguzzi, editors, Proceedings of the 36th International Conference on Logic Programming (Technical Communications), pages 258–260, Waterloo, Australia, 2020.

- Damiano Azzolini, Fabrizio Riguzzi, Elena Bellodi, and Evelina Lamma. A Probabilistic Logic Model of Lightning Network. Business Information Systems Workshops. In press.

3.1 How to Read this Thesis

We tried to keep this dissertation as compact as possible, while, at the same time, making it self-contained. The two streams of work, one involving extensions of PLP, and one involving applications of PLP, are bridged together by the chapters regarding the possible languages and inference methods for PLP. Figure 3.1 shows the dependencies across the several chapters. It should be read from left to right, where an arrow from a chapter A directed to a chapter B means that A is necessary to understand B.

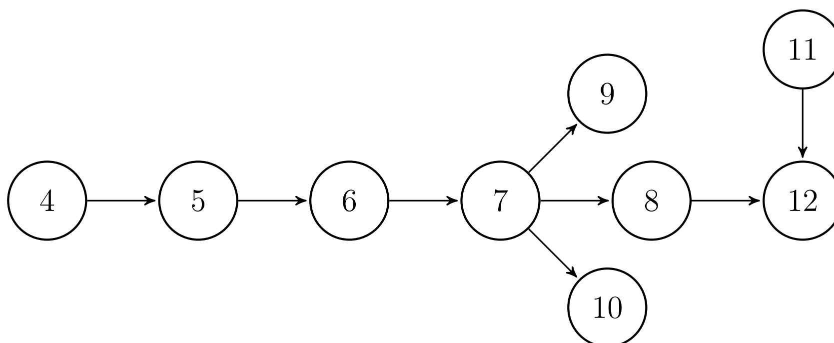


Figure 3.1: Chapter dependency graph.

Part II

Background

Chapter 4

Probability Theory and Set Theory

The following sections provide some background knowledge regarding basic concepts needed to understand this thesis. In particular, in Section 4.1 we review some basic definitions regarding set theory, such as the ones of *lattice* and *order*, in Section 4.2 we introduce the key concepts of Kolmogorov probability theory, such as *sample space* and *probability space*, and in Section 4.3 we recall the definitions of ordinal numbers, mappings, and fixpoints.

4.1 Set Theory

Defined as one of the cornerstones of mathematics, set theory provides formal tools to reason about sets. Informally, a set is a collection of objects. If an object o belongs to a set S, we call it an *element* (or member) of the set and denote it with $o \in S$. The classical notation uses curly brackets $\{\}$ to enclose the elements that constitute a set, separated by commas. For example, a set S that contains the elements 1, 2, and 3 is represented as $\{1, 2, 3\}$. Recall that a function is *injective* (or *one-to-one*) if it maps distinct values to distinct elements. That is, every element of the codomain of the function is the image of at most one element of the domain: given a function f, if $f(a) = f(b)$, then $a = b$.

A set A is equipotent to a set B if there exists a one-to-one function from the former to the latter. If a set is equipotent to the set of natural numbers $\mathbb{N}$ it is

called *denumerable*. A set A is *countable* if there is a one-to-one correspondence between A and a subset B of $\mathbb{N}$. If $B = \{1, 2, \ldots, n\}$ and A is countable, then A is *finite* with n elements. If the definition of countable does not hold, the set is *uncountable*. We assume that the empty set, represented with $\emptyset$, is finite with 0 elements. The powerset of a set A, $\mathcal{P}(A)$, is the set of all subsets of A, including the empty set. For example, if $A = \{1, 2\}$, $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. The cardinality (number of elements) of the powerset of a set A is $2^{|A|}$, where $|A|$ is the cardinality of A.

Given two sets A and B, we can define some relations between these. If all the elements of A are also elements of B, then A is a *subset* of B and it is indicated with $A \subseteq B$. If A is a subset of B and A does not contain the same elements of B, we use the notation $A \subset B$, and A is called *proper subset* of B. Consequently, B is a superset of A, denoted with $B \supseteq A$ (or $B \supset A$ for *proper superset*). For example, the following relations hold:

$$
\begin{aligned}
&\{1, 2, 3\} \subseteq \{1, 2, 3\} \\
&\{1, 2, 3\} \supseteq \{1, 2, 3\} \\
&\{1, 2\} \subset \{1, 2, 3\} \\
&\{1, 2, 3\} \supset \{1, 2\} \\
&\{1, 2\} \subseteq \{1, 2, 3\} \\
&\{1, 2, 3\} \supseteq \{1, 2\}
\end{aligned}
$$

Clearly, given two sets A and B, if both $A \subseteq B$ and $A \supseteq B$ are true, then $A = B$.

Several basic operations can be defined between sets.

- *Union* ($\cup$): $x \in A \cup B \iff x \in A$ or $x \in B$. For example, $\{1, 2\} \cup \{3, 4\} = \{1, 2, 3, 4\}$, $\{1, 2\} \cup \{2, 4\} = \{1, 2, 4\}$. Union is commutative: $A \cup B = B \cup A$.

- *Intersection* ($\cap$): $x \in A \cap B \iff x \in A$ and $x \in B$. For example: $\{1, 2\} \cap \{3, 4\} = \emptyset$, $\{1, 2\} \cap \{2, 4\} = \{2\}$. Intersection is commutative $A \cap B = B \cap A$.

- *Difference* ($\backslash$): $A \backslash B = \{x \in A, x \notin B\}$. For example: $\{1, 2\} \backslash \{3, 4\} =$

$\{1, 2\}$, $\{1, 2, 3, 4\} \backslash \{2, 4\} = \{1, 3\}$. If $A \neq B$ then $A \backslash B \neq B \backslash A$ (set difference is not commutative).

If we consider a reference space set S and a set A subset of S then the complement of A with respect to S is $S \backslash A$ and it is indicated with A^C. For example, if $S = \{1, 2, 3\}$ and $A = \{1, 2\}$, $A^c = \{3\}$.

An *order* on a set A is a binary relation $\leq$ that is, $\forall a, b, c \in A$:

- *antisymmetric*: $a \leq b,\ b \leq a \implies a = b$;
- *reflexive*: $a \leq a$;
- *transitive*: $a \leq b,\ b \leq c \implies a \leq c$.

If the three previous properties hold, the set is *partially ordered* (or simply ordered). If, in addition, the relation has also the following property

- *strongly connected*: $a \leq b$ or $b \leq a$

then, it is a *total order* and the set is *totally ordered*. Given an ordered set B, the upper (lower) bound of a set $A \subseteq B$ is an element $b \in B$ such that $\forall a \in A$, $a \leq b$ ($a \geq b$). If $b \leq b'$ ($b \geq b'$) forall upper (lower) bounds b', then b is the *least upper bound* (*greatest lower bound*), usually abbreviated with lub (glb). If lub or glb (or both) exist, they are unique, and they are also commonly called supremum and infinum respectively.

A *complete lattice* is a partially ordered set A where, $\forall S \subseteq A$, S has both a lub and a glb. A partially ordered set P has a *top element* $\top = \text{lub}(A)$. Similarly, the *bottom element* $\perp$ is $\text{glb}(A)$. For a more in-depth treatment of the topic see [57].

4.2 Probability Theory

In this section, we review some of the basic concepts regarding probability theory, and, in particular, Kolmogorov probability theory. For a complete overview of the field please consult [47].

A *random* experiment is an experiment whose outcome cannot be predicted. Each random experiment has an associated *sample space*, usually indicated with W, that denotes the set of possible *outcomes* (also called *events*).

For example, if we toss a single coin which can land heads or tails, the sample space is $W_1^c = \{h, t\}$. If we toss two coins we need to consider all the possible combinations of outcomes, so $W_2^c = \{(h,h),(h,t),(t,h),(t,t)\}$. More generally, if we toss an infinite number of coins, $W_\infty^c = \{(o_1, o_2, \dots) \mid \forall i \in 1, 2, \dots, o_i \in \{h, t\}\}$. For an experiment with n trials with two possible outcomes each, $|W| = 2^n$. As another example, consider the toss of a single die with six faces. The sample space for a single toss is $W = \{1, 2, 3, 4, 5, 6\}$.

We now provide some definitions needed to introduce the Kolmogorov probability theory.

Let W be a set.

Definition 1 (Algebra and σ-algebra)**.** *A non empty set Ω of subsets of W is an algebra if:*

- *Ω is closed under complementation: $\forall \omega \in \Omega, \omega^c \in \Omega$ and*
- *$\forall \omega_1, \omega_2 \in \Omega$, $\omega_1 \cup \omega_2 \in \Omega$.*

Moreover, if the following also holds

- *Ω is closed under countable union: $\forall \omega_i \in \Omega$, $\bigcup_{n=1}^{\infty} \omega_i \in \Omega$*

Ω is a σ-algebra on W.

The elements of the σ-algebra Ω are called *measurable sets* or events, and (W, Ω) is a *measurable space.* When W is finite, we can always find a σ-algebra by considering the powerset of W, namely $\Omega = \mathcal{P}(W)$. However, there exists also σ-algebras that does not coincide with the powerset. For example, if $W = \{1, 2, 3, 4\}$, a possible σ-algebra is $\Omega = \{\emptyset, \{1, 2\}, \{3, 4\}, \{1, 2, 3, 4\}\}$, which is different than the powerset of W but still satisfies the three previously listed requirements. Consider again the coin toss experiment introduced at the beginning of this section: we can consider the set of events $\Omega^c = \mathcal{P}(W_1^c)$ and $\{t\}$ as an event associated to the outcome tails.

Definition 2 (Minimal σ-algebra)**.** *Let A be an arbitrary collection of non-empty subsets of W. The intersection of all σ-algebras containing the elements of A, denoted with $\sigma(A)$, is called the σ-algebra generated by A or the minimal σ-algebra containing A. $\sigma(A)$ always exists and it is unique [47].*

Definition 3 (Probability measure)**.** *Given a measurable space* (W, Ω)*, a probability measure is a function* $\mu : \Omega \to \mathbb{R}$ *that satisfies the three following axioms (called Kolmogorov axioms):*

1. *Non negative:* $\forall \omega \in \Omega$*,* $\mu(\omega) \geq 0$*.*
2. $\mu(W) = 1$ *(the measure of the sample space is 1).*
3. *Countably additive: if* $S = \{\omega_1, \omega_2, \dots\} \subseteq \Omega$ *is a countable collection of pairwise disjoint sets, then* $\mu(\bigcup_{\omega_i \in S} \omega_i) = \sum_{\omega_i \in S} \mu(\omega_i)$*.*

In particular, 1) and 2) state that the probability of an event must be in the range $[0, 1]$, and 3) imposes that the probability of the union of disjoint events must be equal to the sum of the probability of every single event. The tuple (W, Σ, μ) is called *probability space*. If we consider again the toss of a coin, (W_1^c, Ω^c, μ^c) with $\mu^c(\emptyset) = 0$, $\mu^c(\{h\}) = 0.5$, $\mu^c(\{t\}) = 0.5$, and $\mu^c(\{h, t\}) = \mu(W) = 1$ is a probability space. Given two events A and B, we can define the *conditional* probability (the probability of A given that B happened) as:

$$\mu(A \mid B) = \frac{\mu(A \cap B)}{\mu(B)}$$

with the constraint that $\mu(B) > 0$.

We conclude this part with the definition of product σ-algebra, that will be needed later.

Definition 4 (Product σ-algebra)**.** *Given two measurable spaces* (W_1, Ω_1) *and* (W_2, Ω_2)*, the product* σ*-algebra* $\Omega_1 \otimes \Omega_2$ *is defined as* $\Omega_1 \otimes \Omega_2 = \sigma(\{\omega_1 \times \omega_2 \mid \omega_1 \in \Omega_1, \omega_2 \in \Omega_2\})$*.* $\Omega_1 \otimes \Omega_2$ *is different from the Cartesian product* $\Omega_1 \times \Omega_2$ *because it is the minimal* σ*-algebra generated by all the possible couples of elements from* Ω_1 *and* Ω_2*.* $\Omega_1 \otimes \Omega_2$ *is also called a tensor product.*

4.2.1 Random Variables

Another important concept is the one of random variable. To define it, we need to introduce the definition of measurable function:

Definition 5 (Measurable function)**.** *Given a probability space* (W, Ω, μ) *and a measurable space* (S, Σ)*, a function* $X : W \to S$ *is measurable if* $\forall \sigma \in \Sigma$*,* $X^{-1}(\sigma) = \{w \in W \mid X(w) \in \sigma\} \in \Omega$*.*

Definition 6 (Random variable, r.v.). *Let (W, Ω, μ) be a probability space and (S, Σ) be a measurable space, a measurable function $X : W \to S$ is a random variable (usually abbreviated with r.v.) and the elements of S are called values of X. $\forall \sigma \in \Sigma$, we indicate with $P(X \in \sigma)$ the probability that a random variable X has value in σ, i.e., $\mu(X^{-1}(\sigma))$. The structure of S defines the type of the random variable: if S is finite or countable, X is a discrete random variable; if S is uncountable, X is a continuous random variable.*

For a discrete random variable X_d, we define the *probability mass* as $P(X_d \in x)\ \forall x \in S$, often abbreviated with $P(X_d = x)$, $P(x)$, or $p_{X_d}(x)$. For a continuous random variable $X_c : (W, \Omega) \to (\mathbb{R}, B)$, we define the *probability density* as the function $f_{X_c}(x)$ such that for any measurable set $A \subseteq B$, $P(X \in A) = \int_A f_{X_c}(x)\ dx$.

In other words, a discrete random variable is identified by its probability mass function $p_{X_d}(x)$ that associates a probability value to every outcome. If we have a discrete random variable with n possible outcomes, then $\sum_{i=1}^{n} p_{X_d}(x_i) = 1$ and $p(x_i) \geq 0\ \forall i = 1, \dots, n$. Similarly, a continuous random variable is identified by a probability density function $f_{X_c}(x)$ with the properties that $f_{X_c}(x) \geq 0$ and $\int_{-\infty}^{\infty} f_{X_c}(x)\ dx = 1$. The *cumulative distribution* of a continuous random variable X is defined as $F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_{X_c}(t)\ dt$. The *expected value* of a discrete random variable X_d with n possible outcomes $\{x_1, \dots, x_n\}$, denoted with $E(X_d)$ or $E[X_d]$, is:

$$E(X_d) = \sum_{i=1}^{n} x_i \cdot p_i$$

where $p_i = p_{X_d}(x_i)$. In a similar fashion, for a continuous random variable X_c, the expected value is defined as:

$$E(X_c) = \int_{-\infty}^{\infty} x \cdot f_{X_c}(x)\ dx.$$

If multiple random variables interact, we need to utilize the *joint* probability distribution, which is the probability distribution that describes their joint behavior. We consider here, for simplicity, only two continuous random variables, X and Y (this concept can be extended to an arbitrary number of r.v.s, to discrete r.v.s, and also to a mixture of both discrete and continuous

r.v.s). If the joint probability density is defined as $f_{X,Y}(x,y)$, the *marginal* probability density function that describes individually one of the two random variables (for example X) can be obtained from the joint probability density by marginalization:

$$f_X(x) = \int f_{X,Y}(x,y)\ dy$$

where the integral is on the domain of y. However, the previous integral is tractable only for small cases and simple distributions. If we have two discrete random variables, we need to replace the integral with a summation. Finally, in the case of a mixture of discrete and continuous random variables, we have a mixed joint density, and we need to consider both summations and integrations.

If both X and Y are continuous, we define the joint cumulative distribution as $F_{X,Y}(x,y) = P(X \leq x, Y \leq y)$; in the case both are discrete, the joint mass function is defined as $p_{X,Y}(x,y) = P(X = x, Y = y)$.

Two variables are *independent* if the knowledge of the outcome of one of the two does not influence the outcome of the other. This is an important property since, for independent variables, $F_{X,Y}(x,y) = F_X(x) \cdot F_Y(y)$, and $P(X = x, Y = y) = P(X = x) \cdot P(Y = y)$. This factorization greatly simplifies computations, and it is usually considered in Probabilistic Logic Programming.

There are several well-known distributions for random variables. Here, we introduce only three of them, two discrete (Bernoulli and categorical) and one continuous (normal).

The *Bernoulli* distribution is a discrete probability distribution with a parameter $p \in [0,1]$ representing a random variable X that takes value 1 (or true, or, in other words, the trial succeeds) with probability p and 0 (or false, or the trial fails) with probability $1 - p$: $P(X = 1) = p$, $P(X = 0) = 1 - p$. If we consider the toss of a fair coin, its outcome can be modelled with a Bernoulli distribution with $p = 0.5$ (the probability to land either heads or tails is the same). If the coin is not fair, $p \neq 0.5$. To indicate a variable X that follows a particular distribution we use the symbol $\sim$. For example, to indicate that X follows a Bernoulli distribution with parameter p we write $X \sim \text{Bernoulli}(p)$. If the number of possible outcomes is greater than two, we need to consider a *categorical* distribution: each outcome i has a probability p_i, and $\sum_i p_i = 1$. For example, the toss of a die can be modelled with a categorical distribution:

if it is fair, each one of the possible six outcomes has probability 1/6.

As an example of continuous distribution that is used in many scenarios, we introduce the *normal* distribution, parameterized by two values, μ for the mean and σ^2 for the variance, that is usually indicated with $\mathcal{N}(\mu, \sigma^2)$. The probability density for this distribution is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

If $\mu = 0$ and $\sigma^2 = 1$, the distribution is a *standard* normal.

4.3 Ordinal Numbers, Mappings, and Fixpoints

In this section, we recall some of the concept of ordinal numbers, mappings, and fixpoints needed to understand the proof of the theorems discussed later.

4.3.1 Ordinal Numbers

In set theory, the notion of *ordinal* numbers extends the one of natural numbers. We indicate with Ω the set of ordinal numbers[1]. The elements of Ω are called *ordinals.* Ω is well-ordered, meaning that it is totally ordered and every subset of it has a smallest element. In this case, the smallest element is 0. If we consider two ordinals, α and β, we say that α is *predecessor* of β (or β is the *successor* of α) if $\alpha < \beta$. If α is the largest ordinal smaller than β, α is the *immediate predecessor* of β. Similarly, the *immediate successor* of an ordinal α is the smallest ordinal larger than it, and is indicated with $\alpha + 1$. Ordinals that have predecessors but no immediate predecessor are *limit ordinals*, and every ordinal has an immediate successor named *successor ordinal.* That is, ordinal numbers can be of two types: *limit ordinals* or *successor ordinals.* The first elements of Ω are 0 (empty set), 1 (set with one element, $\{0\}$), 2 (set with two elements, $\{0,1\}$), ... After all the natural numbers there is ω, the set of all finite ordinals ($\{0,1,2,\ldots\}$). The successors of ω are $\omega + 1$ ($\{0,1,2,\ldots,\omega\}$), $\omega + 2$, and so on. For ordinal numbers, the concept of *transfinite sequence* generalizes the concept of sequence. Similarly, the concept of induction for

[1] There is a slight abuse of notation: in Section 4.2 we used Ω to indicate the event space. Here, we use the same greek letter to indicate the set of ordinal numbers.

ordinal numbers goes under the name of *transfinite induction*: if a property P is defined for an ordinal α ($P(\alpha)$) and it holds whenever P holds for another ordinal β ($P(\beta)$) with $\beta < \alpha$, then $P(\alpha)$ is also true. Proofs by transfinite induction usually consist of three steps:

- The property P is proved for the base case $P(0)$.
- P is proved for successor ordinal $\alpha + 1$ by proving that $P(\alpha + 1)$ follows from $P(\alpha)$.
- P is proved for limit ordinals γ by proving that $P(\gamma)$ follows $\forall \beta$ from $P(\beta)$ with $\gamma > \beta$.

For a more in-depth treatment of the topic of ordinal numbers see [51].

4.3.2 Mappings and Fixpoints

We now focus on the concepts of mapping and fixpoint.

Given a lattice A, a *mapping* is a function $f : A \to A$, that is, a function that maps the elements of its domain (a lattice) to itself. Sometimes, it is also called *operator*. A mapping is *monotone* if $f(x) \leq f(y)$ $\forall x, y \in A$ such that $x \leq y$. If $a \in A$ and $f(a) = a$, then a is a *fixpoint*. $a \in A$ is the *least fixpoint* if a is a fixpoint and, for all the other fixpoints b_i, $a \leq b_i$ holds. Similarly, $a \in f$ is the *greatest fixpoint* if a is a fixpoint and, for all the other fixpoints $b_i \in A$, $a \geq b_i$ holds.

Consider again a lattice A and a monotonic mapping f. We inductively define *increasing ordinal* powers of f as:

- $f \uparrow 0 = \bot$.
- If α is a successor ordinal, $f \uparrow (\alpha + 1) = f(f \uparrow \alpha)$.
- If α is a limit ordinal, $f \uparrow \alpha = \text{lub}(\{f \uparrow \beta \mid \beta < \alpha\})$.

Similarly, we inductively define *decreasing ordinal powers* of f as:

- $f \downarrow 0 = \top$.
- If α is a successor ordinal, $f \downarrow \alpha = f(f \downarrow (\alpha - 1))$.

- If α is a limit ordinal, $f \downarrow \alpha = \text{glb}(\{f \downarrow \beta \mid \beta < \alpha\})$.

Thanks to the Knaster-Tarski theorem (see [77]), if A is a complete lattice and f is a monotonic mapping, the set of fixpoints of f in A is also a lattice. Moreover, f has a least fixpoint $\text{lfp}(f)$ and a greatest fixpoint $\text{gfp}(f)$. For further details consult [65, 77].

Chapter 5

Logic and Logic Programming

In this chapter, we introduce some concepts of Logic and Logic Programming, by considering both syntax and semantics. In particular, Section 5.1 reviews basic definitions regarding Logic. Logic Programming is extensively analysed in Section 5.2. Finally, Section 5.3 reviews Abductive Logic Programming, an extension of Logic Programming that manages incompleteness in the data.

5.1 Propositional and First Order Logic

Logic is the science that studies reasoning methods and conditions that makes the process of reasoning correct. One of the goals of this discipline is to develop rules to distinguish correct statements from fallacious ones. There are different types of Logic: we start with *Propositional Logic* (PL).

5.1.1 Propositional Logic

The main elements of Propositional Logic (and of all the other types of logics in general) are formulas (or formulae). In PL, formulas are constituted by atomic propositions, i.e., propositions that cannot be further simplified, linked together by logical connectives. Atomic propositions are indicated with symbols such as $a, b, \ldots$. The *truth value* of an atomic proposition (or a logic formula) is the value that it assumes when evaluated, and can be either true or false, indicated respectively with T (or 1, or $\top$) and F (or 0, or $\bot$). Note that we only define the truth value of a sentence, not its meaning. There are

several types of logics that admit more than two truth values, but we will not consider them here.

We can chain multiple propositions together through logical connectives. These are:

- *Not* (negation) indicated with $\neg$, that flips the truth value of a formula. If we have a formula a which is true, $\neg a$ is false, and vice versa. This is the only unary operator, that is, it is applied to a single formula. $\neg\neg a$ has the same truth value of a.

- *And* (conjunction) indicated with $\wedge$. If we have two formulas a and b, $a \wedge b$ is true only if both a and b are true; otherwise, the conjunction is false.

- *Or* (disjunction) indicated with $\vee$. If we have two formulas a and b, $a \vee b$ is false only if both a and b are false; otherwise, the disjunction is true.

- *Material implication* denoted with $\rightarrow$. If we have two formulas a and b, $a \rightarrow b$ is false only if a is true and b is false; otherwise, it is true. It can be read as: "if ... then".

- *Material biconditional* denoted with $\leftrightarrow$. If we have two formulas a and b, $a \leftrightarrow b$ is true only if a and b are both true or false simultaneously; otherwise, it is false. It can be read as "... if and only if ...".

To specify priorities, formulas can be enclosed in parentheses. Clearly, to be well-formed, the number of parentheses in a formula must be balanced (the number of open parentheses must equal the number of closed parentheses). By default, we assume the following operator precedence (from higher to lower): $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$. Some examples of well-formed formulas are:

- $a \vee b$.

- $c \vee b \wedge d$.

- $\neg a \rightarrow c \wedge (d \vee b)$.

Considering the precedence rule specified above, the following two formulas are equivalent: $\neg a \rightarrow b \vee c \rightarrow a \vee b \wedge c$, $(\neg a) \rightarrow ((b \vee c) \rightarrow (a \vee (b \wedge c)))$. However, the second is usually preferred due to higher interpretability, since priorities are clearly highlighted by the parentheses.

5.1.2 First Order Logic

One of the main limitations of PL is that it is not expressive enough, since it does not allow reasoning over non-logical objects. We now consider one of its extensions, First Order Logic (FOL), also named Predicate Logic or First Order Predicate Calculus, that allows the usage of quantifiers and variables. We now introduce the syntactic constructs (or, more formally, the *alphabet*) allowed by FOL. These are:

- *Constants*, that represent the known entities of the domains, starting with a lowercase letter, such as "a", "home", "learning". Numbers are considered constants.

- *Variables*, starting with an uppercase letter and referring to the objects of the domain. For example, "X", "Y", "Thesis".

- *Function symbols*, starting with lowercase letters, mapping n (> 0) objects to another object. n is called *arity*. Function symbols are usually represented with the syntax *functor/arity*, where *functor* is the name of the mapping they define. For example $f(a, B)$ represents a function called f mapping a constant a and a variable B to another variable. We use $f/2$ to represent this function. Similarly, $mother(bob)$ represents a function called $mother$ from a constant bob to another object, and can be compactly referred as $mother/1$.

- *Predicate symbols*, starting, as for function symbols, with a lowercase letter and mapping n (> 0) objects to truth values (true or false). The notation is the same used for function symbols (*functor/arity*). For example: $father(marc, bob)$, $faster(hare, tortoise)$. The difference between function and predicate symbols is that the former denotes functions while the latter relations.

- *Logical connectives*, the same of PL.

- *Auxiliary punctuation*: as for PL, it is possible to use, for instance, parentheses, to specify priorities.

- *Quantifiers*: universal quantifier, represented with $\forall$ (for all), and existential quantifier, represented with $\exists$ (there exists). Both have the same priority as negation (the highest).

Starting from the previous alphabet, we can introduce some new constructs:

- *Term*: a variable, a constant, or a function symbol applied to n terms.
- *Atom* (or atomic formula): predicate symbol applied to n terms.
- *Literal*: an atom a or its negation (denoted $\sim a$ or $\neg a$).

Well-Formed Formulas are syntactically correct formulas. They are built starting from atomic formulas combined with logical connectives and quantifiers. More formally, we can provide an inductive definition of well-formed formula:

- Atomic formulas, true, and false are formulas.
- If A and B are formulas, the following are still formulas: $\neg A$, $A \vee B$, $A \wedge B$, $A \rightarrow B$ (and so $A \leftarrow B$), $A \leftrightarrow B$.
- If A is a formula and X is a variable, $\forall X A$, and $\exists X A$ are formulas.

A quantifier is applied to the formula that immediately follows it, and this defines its *scope*. As usual, we can utilize parentheses to modify the scope of a quantifier, by joining and combining two or more formulas. A quantifier is applied to a formula with variables. According to the variable appearing in the quantifier, a variable can either be *bound* or *free*. A variable is free if it does not appear in the scope of any quantifier, otherwise, it is bound. For example, in $\forall X\ p(X, Y)$, Y is free while X is bound.

A *clause* in FOL is a formula of the form

$$A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_m$$

where A_is and B_is are atoms and all the variables are universally quantified from the outside (the explicit introduction of the quantifier is often omitted). The previous formula can be rearranged to remove the negation symbol, as follows:

$$A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m.$$

In this new notation, that we will adopt through the thesis, $A_1 \vee \cdots \vee A_n$ is called *head* while $B_1 \wedge \cdots \wedge B_m$ is called *body* with the meaning "if the body is true then the head is also true". Furthermore, usually $\vee$ is replaced by semicolon (;) and $\wedge$ by commas (,). For example, the following two clauses

$$\begin{aligned} &\mathit{siblings}(Xa, Xb) \leftarrow \mathit{father}(Xa, Xc), \mathit{father}(Xb, Xc) \\ &\mathit{siblings}(Xa, Xb) \leftarrow \mathit{mother}(Xa, Xd), \mathit{mother}(Xb, Xd) \end{aligned}$$

state that Xa and Xb are siblings ($\mathit{siblings}(Xa, Xb)$) if ($\leftarrow$) they have the same father Xc (first clause) or the same mother Xd (second clause). The two previous clauses are called *definite* since they have only one positive literal. If the number of positive literals is greater than one, the clause is *disjunctive.* A clause without negative literals (the body is empty) is called *fact* and represents an information that is always true. For example

$$\mathit{fly}(\mathit{hoopoe}) \leftarrow \mathit{true}$$

is a fact and can be represented more compactly as

$$\mathit{fly}(\mathit{hoopoe})$$

removing *true* in the body. Conversely, if the head is empty, the clause is a *denial*:

$$\mathit{false} \leftarrow \mathit{fly}(\mathit{hoopoe}), \neg\mathit{fly}(\mathit{hoopoe})$$

or, equivalently

$$\leftarrow \mathit{fly}(\mathit{hoopoe}), \neg\mathit{fly}(\mathit{hoopoe}).$$

Horn clauses are of particular interest for Logic Programming and are either definite clauses (and also facts) or denials. A set of clauses defines a *theory.*

In general, we call *expression* a literal, a term, or a clause. An expression is *ground* if it does not contain variables. The process of replacing variables with constants is called *substitution.* A substitution applied to an expression E consists in replacing simultaneously variables in E with terms, and it is usually indicated with $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$, where the V_is are variables in E. The result of the application of a substitution is a new expression $E\theta$

called *instance* of E. For example, given the expression $E = \mathit{father}(A, B)$, the substitution $\theta = \{A/\mathit{alfred}, B/\mathit{bruna}\}$ applied to E creates a new expression $E\theta = \mathit{father}(\mathit{alfred}, \mathit{bruna})$. If a substitution grounds an expression (as θ of the previous example), it is called *grounding*. Grounding is also the name of the process that consists in replacing variables with constants to make a clause or a program ground. Not all variables of an expression must be present in a substitution. If the substitution is a permutation of the set of variables it is called *renaming*. The substitution is at the heart of the process of *unification*, used to check whether two formulas can be made syntactically equal through a series of substitutions. We call a substitution θ *most general unifier* of two expressions iff (if and only if) for every substitution λ that unifies the two expressions there exists a substitution σ such that $\lambda = \theta\sigma$, that is, θ is the minimal substitution that unifies the two expressions.

The *Herbrand universe* of a theory T is the set of all ground terms that can be obtained by combining the symbols and the constants in T in all possible ways. If there are no function symbols, the Herbrand universe is finite. Similarly, the *Herbrand base* is the set of all ground atoms (atomic formulas) constructed using the symbols in the program. For example, consider the following theory constituted by one fact and one definite clause:

$$\mathit{even}(0)$$
$$\mathit{even}(s(s(X))) \leftarrow \mathit{even}(X)$$

$0, s(0), \ldots$ can be used to denote the element of Peano arithmetic: $s(0)$ represents the successor of the number 0, 1, $s(s(0))$ the successor of $s(0)$, 2, and so on. The theory states that 0 is even and that $s(s(X))$ is even if X is even. There is one constant 0 and a function symbol $s/1$, so the Herbrand universe of the program is infinite and is

$$\{0, s(0), s(s(0)), \ldots\}.$$

Its Herbrand base is as well infinite, since there is one predicate symbol $\mathit{even}/2$, and is

$$\{\mathit{even}(0), \mathit{even}(s(0)), \mathit{even}(s(s(0))), \ldots\}.$$

The *semantics* of a set of FOL formulas, i.e., their meaning, can be provided through *Herbrand interpretations* and *Herbrand models* [105]. A Herbrand interpretation, also called *two-valued interpretation* or simply *interpretation*, is a subset of the Herbrand base and allows to assign a truth value to formulas. Consider and interpretation I:

- A ground atom $p(t_1, \ldots, t_n)$ is true in I iff $p(t_1, \ldots, t_n) \in I$.
- A conjunction of atomic formulas $C = a_1, \ldots, a_n$ is true in I iff $C \subseteq I$.
- A ground clause $h_1; \ldots; h_n \leftarrow b_1, \ldots, b_n$ is true in I iff at least one h_i is true when the body (conjunction of b_i) is true.
- A clause is true in I iff all its ground instances are true in I.
- A set of clauses is true in I iff all its clauses are true in I.

An interpretation I satisfies a set of clauses Σ if Σ is true in I, and it is denoted with $I \models \Sigma$. In this case, I is called *Herbrand model* or simply *model* of the set of clauses. In the special case that all the models for a set of clauses Σ are also models of a single clause C, Σ *logically entails* C and C is a logical consequence of Σ, written $\Sigma \models C$. We say that a set of clauses Σ is *satisfiable* if it has at least one Herbrand model; otherwise, it is *unsatisfiable*. We consider this definition as the semantics for Σ.

An important property of Herbrand models is that, given a set of definite clauses (i.e., a definite program), the intersection of all the Herbrand models of the set is still a Herbrand model for the set, and it is called the *least Herbrand model*. Furthermore, the least Herbrand model for a definite program always exists, is minimal, and it is unique. With the least Herbrand model we can provide a *model-theoretic semantics* of a program as the set of grounds atoms that are logical consequences of P. For example, if we consider the program

$$\mathit{bird}(X) \leftarrow \mathit{penguin}(X)$$
$$\mathit{penguin}(\mathit{coco})$$

it has the least Herbrand model $\{\mathit{penguin}(\mathit{coco}), \mathit{bird}(\mathit{coco})\}$. See [105] for more details.

5.2 Logic Programming

Logic Programming, initially proposed in 1974 [99], has its root in FOL but adopts a different semantics. A *disjunctive logic program* is a set of *clauses* (also called *rules*) of the following form:

$$h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m.$$

where the symbols are interpreted as for FOL. Each h_i is an atom and each b_i is a literal (so negation is allowed). Note that, differently from FOL, clauses end with a full stop. As before, if the head has only one atom ($n = 1$), the clause is a *normal* clause. If the program is composed of only normal clauses, it is termed *normal logic program.* A *definite logic program* is such that, for every clause, $n = 1$ and every b_i is a positive literal (i.e., an atom).

Prolog [53] is the most famous programming language based on logic. It adopts a *proof procedure* called *resolution* to check whether a formula can be proved starting from a theory. A key feature of resolution is that it can be easily automated. Prolog adopts a particular form of resolution, called *SLD resolution* (that stands for "linear resolution with selection function for definite logic programs") that we now describe. In the following, we will use `typewriter` font to indicate Prolog clauses that are well-formed and can be executed as they are in a Prolog engine. In this case, the symbol $\leftarrow$ is replaced by `:-`. SLD resolution starts from a *goal* o *query*

$$a_1, a_2, \ldots, a_n.$$

(a conjunction of atoms) and iteratively selects a subgoal and replaces it with the body of a clause in the program which has the head unifiable with the subgoal itself. For example, if we select the goal a_1 from the previous example and there exists a rule

$$b_0 \leftarrow b_1, \ldots, b_m.$$

where a_1 can be unified with b_0 through a substitution θ, the new goal to prove becomes

$$(b_1, \ldots, b_m, a_2, \ldots, a_n)\theta.$$

There can be two reasons for stopping the resolution process: the goal is empty (in this case we have a *refutation*) and the query succeeds, or there are no more applicable resolutions (i.e., clauses that match the current subgoal to prove), and the goal fails. Resolution has been proved *sound* (answers to queries are logical consequences of the program) and *complete* (every possible answer can be derived) for definite logic programs.

In the version of SLD resolution adopted by Prolog, the next subgoal to prove is always the leftmost one (a_1 in the previous example). When there are multiple clauses with the head that can be unified with the current goal, the first in order of appearance starting from the top of the program is selected. However, this choice can lead to infinite derivations, so the SLD resolution procedure adopted by Prolog is not complete. To see an example of Prolog SLD resolution, consider the following program, where variables start with uppercase letters[1]

```
1 reach(X,X).
2 reach(X,Y):- connected(X,Z), reach(Z,Y).
3
4 connected(a,b).
5 connected(b,c).
6 connected(a,c).
```

The first two clauses define the *predicate* `reach/2` and can be read as follows: from `X` we can reach `X` (this is always true, since we are already in `X`); otherwise, we can reach `Y` starting from `X` if `X` is connected to an intermediate location (`Z`) from which we can reach `Y`. The last three facts define the available connections. The SLD tree for the query `reach(a,c)` is shown in Figure 5.1, where substitutions are displayed on edges between two different levels of the tree, □ denotes successful proofs, and `fail` denotes proof that failed. When the goal does not contain variables, a Prolog engine can answer with `true`, if the resolution succeeds, or `false`, if it fails. If the goal contains variable, a Prolog engine will either succeed returning a substitution or fail with answer `false`.

To better see why the Prolog SLD resolution is not complete, consider the

[1]Not all variables in Prolog start with an uppercase letter. For example, the *anonymous* variable is represented by an underscore (`_`) and each occurrence of this denotes a different variable.

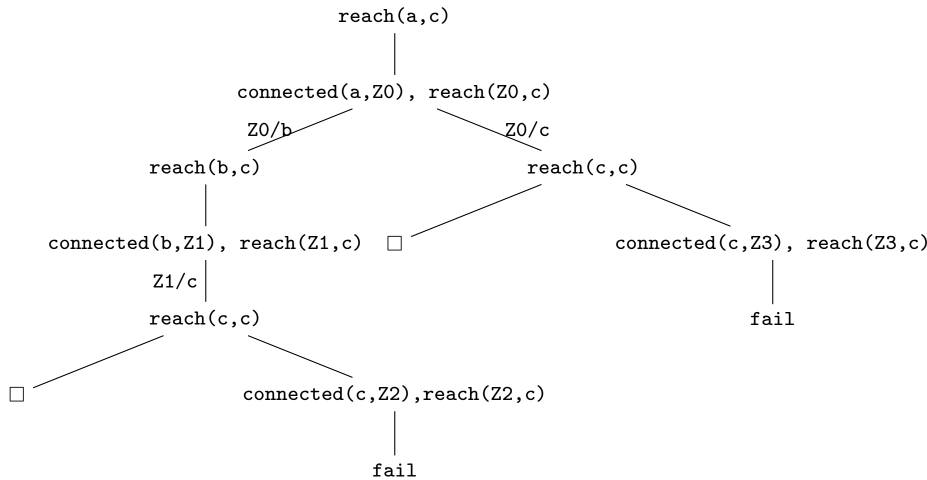


Figure 5.1: SLD tree for the query `reach(a,c)`.

following simple program:

```
1 connected(A,B):- connected(B,A).
2 connected(a,b).
```

Here, the query `connected(a,Y)` does not terminate: both the clause and the fact can be unified with the goal, but Prolog chooses the first one in order of appearance, so the clause, and the substitution `{A/a,B/Y}` is applied. The new goal to prove is then `connected(Y,a)`. The clause is selected once again with the subsection `{A/Y,B/a}`, and the new goal is `connected(a,Y)`, falling in an infinite loop (the subgoal to prove is equal to the main goal). However, if we swap the clause with the fact, we get an infinite number of solutions where `Y` is always replaced with `b`. Thus, in Prolog, the order of the clauses matters. Moreover, Logic Programming semantics adopt the *closed world assumption*: everything that cannot be proved is considered false.

There are several Prolog interpreters, among them SWI-Prolog [171], YAP-Prolog [145], and XSB-Prolog [159]. In this thesis, we always consider SWI-Prolog, unless differently specified.

In Section 5.1.2 we stated that the least Herbrand model always exists for programs composed of only definite clauses (with exactly one head atom). The least Herbrand model can be characterized by a fixpoint operator called *immediate consequence operator*, that maps subset of atoms to subset of atoms

for a definite program P. Consider a set of definite clauses P and a Herbrand interpretation I. The immediate consequence operator $T_p(I)$ is defined as follows:

$$T_p(I) = \{A \in B_P \mid A \leftarrow A_1, \ldots, A_n \text{ is a ground instance}$$
$$\text{of a rule in P and } A_1, \ldots, A_n \in I\}.$$

Since we are considering definite programs, the T_P operator is *monotonic*, meaning that:

$$I_1 \subseteq I_2 \implies T_p(I_1) \subseteq T_p(I_2).$$

Furthermore, it has a least (unique and minimal) fixpoint which is $T_p \uparrow^{\omega}$. For example, if we consider the program:

```
1 f(b).
2 f(X) :- g(X,Y),f(Y).
3 g(a,b).
```

we get:

$$T_p \uparrow 0 = \emptyset$$
$$T_p \uparrow 1 = \{g(a,b), f(b)\}$$
$$T_p \uparrow 2 = \{g(a,b), f(b), f(a)\}$$
$$T_p \uparrow 3 = \{g(a,b), f(b), f(a)\}$$
$$\vdots$$
$$T_p \uparrow \omega = \{g(a,b), f(b), f(a)\}$$

Thus, the least fixpoint is $\{g(a,b), f(b), f(a)\}$. Similarly, for the Peano arithmetic program shown before, reported here with Prolog syntax for clarity,

```
1 even(0).
2 even(s(s(X))) :- even(X).
```

we get:

$$T_p \uparrow 0 = \emptyset$$
$$T_p \uparrow 1 = \{even(0)\}$$
$$T_p \uparrow 2 = \{even(s(s(0))), even(0)\}$$
$$T_p \uparrow 3 = \{even(s(s(s(s(0))))), even(s(s(0))), even(0)\}$$
$$\vdots$$
$$T_p \uparrow \omega = \{even(s^i(0)) \mid 2i \in \mathbb{N}\}$$

In these examples, negation is not considered, since it requires a more complex semantics that we describe in the next subsection.

5.2.1 Semantics for Programs with Negation

Normal logic programs allow negative literals in the body of clauses. In this case, *Negation as Failure* (or default negation) is considered, where `not p` states that `p` is not provable. To manage negation, SLD resolution is extended to SLDNF resolution (SLD with Negation as Failure). In short, when, during the resolution, a negated literal is encountered, the positive version of it is proved: if it is successful, the proof of its negated version fails and the resolution fails as well on this branch; if the proof of the positive literal fails, the negated version succeeds, and the resolution continues. Negation in Prolog is expressed using the operator \+. For example, consider the following simple Prolog program:

```
1 q(1).
2 p(X) :- \+ q(X).
```

If the query is `p(1)`, we get `false` as answer from a Prolog engine, since the subgoal `q(1)` is true. Similarly, `p(2)` succeeds since `q(2)` is false (closed world assumption: what is not explicitly stated is false). However, there are some important points to consider: the negated goal to prove must be ground, that is, it must not contain variables, otherwise, the resolution *flounders*.

Prolog SLDNF follows Clark's completion semantics [52]. However, there

are alternative semantics to handle negations, such as Stable model semantics [68] and Well-Founded semantics (WFS) [131, 165]. Here, we consider only the last one.

We now introduce some additional concepts needed to formally define the Well-Founded semantics. A *two-valued* interpretation for a program P with Herbrand base B_p, $I \subseteq B_P$, represents the set of true and false atoms: if, for an atom a, $a \in I$, a is true; if $a \notin I$, a is false. The set of two-valued interpretations for a program P, Int_2^P, forms a complete lattice where the partial order $\leq$ is defined by the subset relation $\subseteq$. The bottom element of Int_2^P is $\emptyset$ and the top element is B_P. A *three-valued* interpretation $\mathcal{I}$ is a pair $\langle I_T, I_F \rangle$ where $I_T \subseteq B_P$ and $I_F \subseteq B_P$ are sets of respectively true and false atoms. An atom a is true in $\mathcal{I}$ ($\mathcal{I} \models a$) if $a \in I_T$, is false in $\mathcal{I}$ ($\mathcal{I} \models \sim a$) if $a \in I_F$, and is *undefined* in $\mathcal{I}$ if $a \notin I_T$ and $a \notin I_F$. A three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ is consistent if $I_T \cap I_F = \emptyset$. The union and the intersection of two three-valued interpretations $\langle I_T^{'}, I_F^{'} \rangle$ and $\langle I_T^{''}, I_F^{''} \rangle$ are defined respectively as $\langle I_T^{'} \cup I_T^{''}, I_F^{'} \cup I_F^{''} \rangle$ and $\langle I_T^{'} \cap I_T^{''}, I_F^{'} \cap I_F^{''} \rangle$. We represent a three-valued interpretation as a single set of literals composed of $I_T \cup \{\sim a \mid a \in I_F\}$.

The set of three-valued interpretations for a program P, Int_3^P, forms a complete lattice where the partial order $\leq$ is defined as $\langle I_T^{'}, I_F^{'} \rangle \leq \langle I_T^{''}, I_F^{''} \rangle$ if $I_T^{'} \subseteq I_T^{''}$ and $I_F^{'} \subseteq I_F^{''}$. The top and the bottom elements are respectively the pairs $\langle \emptyset, \emptyset \rangle$ and $\langle B_P, B_P \rangle$. While considering a three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$, we define some auxiliary functions: $true(\mathcal{I}) = I_T$, $false(\mathcal{I}) = I_F$, $undef(\mathcal{I}) = B_P \setminus I_{TF}$, where $I_{TF} = I_T \cup I_F$.

The Well-Founded semantics (WFS) [131, 165] assigns a three-valued model to a program. Consequently, the semantics of a program is given by a consistent three valued interpretation. The definition of the WFS provided in [165] is based on the computation of the least fixpoint of an operator composed of two sub-operators. In [131], the same definition is given in terms of an iterated fixpoint. Here we consider this second definition.

Definition 7 ($OpTrue_{\mathcal{I}}^P$ and $OpFalse_{\mathcal{I}}^P$ operators)**.** *For a normal logic program P, two sets Tr and Fa of ground atoms, and a fixed three-valued interpretation $\mathcal{I}$, the operators $OpTrue_{\mathcal{I}}^P : Int_2^P \rightarrow Int_2^P$ and $OpFalse_{\mathcal{I}}^P : Int_2^P \rightarrow Int_2^P$ are defined as follows:*

$OpTrue_{\mathcal{I}}^P(Tr) = \{a \mid$ *a is not true in* $\mathcal{I}$ *and there is a clause* $b \leftarrow l_1, \ldots, l_n$

in P and a grounding substitution θ such that $a = b\theta$ and, for every $1 \leq i \leq n$, either $l_i\theta$ is true in $\mathcal{I}$ or $l_i\theta \in Tr\}$.

$OpFalse^P_{\mathcal{I}}(Fa) =$ *$\{a \mid a$ is not false in $\mathcal{I}$ and for every clause $b \leftarrow l_1, \ldots, l_n$ in P and grounding substitution θ such that $a = b\theta$ there is some i $(1 \leq i \leq n)$ such that $l_i\theta$ is false in $\mathcal{I}$ or $l_i\theta \in Fa\}$.*

That is, $\mathcal{I}$ contains the atoms whose truth values are already known, the operator $OpTrue^P_{\mathcal{I}}(Tr)$ add to $\mathcal{I}$ the new true atoms that can be derived from P knowing $\mathcal{I}$ and true atoms Tr, while $OpFalse^P_{\mathcal{I}}(Fa)$ computes new false atoms in P starting from $\mathcal{I}$ and false atoms Fa. The authors of [131] proved that $OpTrue^P_{\mathcal{I}}$ and $OpFalse^P_{\mathcal{I}}$ are both monotonic, so they both have least fixpoint and a greatest fixpoint. The least fixpoint of $OpTrue^P_{\mathcal{I}}$ contains the new atoms that can be derived from P knowing $\mathcal{I}$. Similarly, the greatest fixpoint of $OpFalse^P_{\mathcal{I}}$ contains the new atoms considered false in P knowing $\mathcal{I}$. Consider now this new operator that iteratively builds three-valued interpretations:

Definition 8 (Iterated fixed point)**.** *For a normal logic program P, let $IFP^P : Int^P_3 \rightarrow Int^P_3$ be defined as*

$$IFP^P(\mathcal{I}) = \mathcal{I} \cup \langle \text{lfp}(OpTrue^P_{\mathcal{I}}), \text{gfp}(OpFalse^P_{\mathcal{I}}) \rangle.$$

It adds to $\mathcal{I}$ the new atoms $\text{lfp}(OpTrue^P_{\mathcal{I}})$ that can be derived from P knowing $\mathcal{I}$, and negated atoms $\text{gfp}(OpFalse^P_{\mathcal{I}})$ considered false in P knowing $\mathcal{I}$. IFP^P is monotonic [131] and so it has a least fixpoint denoted with $\text{lfp}(IFP^P)$. The Well-Founded Model (WFM) of P, $WFM(P)$, is defined as $\text{lfp}(IFP^P)$. Let δ be the smallest ordinal such that $WFM(P) = IFP^P \uparrow \delta$. We call δ the *depth* of P. The *stratum* of atom a is the least ordinal β such that $a \in IFP^P \uparrow \beta$. In this case, a may be either in the true or false component of $IFP^P \uparrow \beta$. Undefined atoms of the WFM are not added to $IFP^P \uparrow \delta$ for any ordinal δ, so they do not belong to any stratum. If $undef(WFM(P)) = \emptyset$, then the WFM is called *total* or *two-valued* and the program is *dynamically stratified.*

To see an example of computation of the WFM, consider the following

program P taken from [136]:

1) $b \leftarrow \sim a$

2) $c \leftarrow \sim b$

3) $c \leftarrow a$

where 1), 2), and 3) are used to denote clause numbers and are not part of the program. By iteratively applying IFP^P operator we get:

$IFP^P \uparrow 0 = \langle \emptyset, \emptyset \rangle$

$IFP^P \uparrow 1 = \langle \emptyset, \{a\} \rangle$

$IFP^P \uparrow 2 = \langle \{b\}, \{a\} \rangle$ clause 1

$IFP^P \uparrow 3 = \langle \{b\}, \{a, c\} \rangle$ clauses 2 and 3

$IFP^P \uparrow 4 = IFP^P \uparrow 3 = WFM(P)$

The depth of P is 3 and its WFM is: $true(WFM(P)) = \{b\}$, $false(WFM(P)) = \{a, c\}$, $undef(WFM(P)) = \emptyset$. The undefined set is empty, so the program is two-valued. Moreover, it is also dynamic stratified.

Finally, we introduce the definition of level mapping and additional types of logic programs, namely *acyclic*, *stratified*, *locally stratified*, and *range restricted*.

Definition 9 (Level mapping [6])**.** *A level mapping for a program P is a function $||: B_P \to \mathbb{N}$ from ground atoms of the Herbrand base to natural numbers. We indicate with $|a|$ the level of $a \in B_p$. If $l = \neg a$ and $a \in B_p$, then $|l| = |a|$.*

Definition 10 (Acyclic program [6])**.** *A program P is called acyclic with respect to a level mapping $||$ if, for every ground instance of a clause $A \leftarrow L_1, \ldots, L_n$ of the program P, $|A| > |L_i|$ $\forall i \in [1, n]$.*

Definition 11 (Locally stratified program [130])**.** *A program P is locally stratified with respect to a level mapping $||$ if, for every ground instance of a clause $A \leftarrow L_1, \ldots, L_n$ of the program P, $|A| > |L_i^-|$, $\forall L_i^-$ negative literal, and $|A| \geq |L_i^+|$, $\forall L_i^+$ positive literal.*

Definition 12 (Stratified program [7])**.** *A normal logic program is stratified if it is locally stratified according to some level mapping and all the ground atoms for the same predicate can be assigned to the same level.*

A locally stratified program P has a total WFM. The program shown before is locally stratified.

Definition 13 (Range restricted program)**.** *A program is range restricted if all the variables appearing in the head of rules also appear in a positive literal of the body.*

If a normal logic program is acyclic, the Well-Founded semantics, the Stable model semantics, and the Clark's completion semantics coincide [133]. If the program is also range restricted, SLDNF resolution is correct, sound, and complete [6]. There are other proof procedures, not reported here, that are also sound and complete for the WFS under some conditions, such as SLG resolution [46].

There exist several extensions to Logic Programming:

- Probabilistic Logic Programming [136], that can manage uncertain data. It will be deeply analysed in Chapter 6.

- Constraint Logic Programming [82], that integrates logic with constraints.

- Inductive Logic Programming [115], focused on learning programs starting from a set of examples.

- Abductive Logic Programming [86], that tackles incompleteness in the data, discussed in the next section.

5.3 Abduction and Abductive Logic Programming

The goal of abduction is guessing missing information from data. Abductive Logic Programming [86, 87] extends Logic Programming and marks some atoms as *abducible*. The goal is to find a subset of abducibles that can explain a query. Furthermore, an abductive logic program has also a set of *integrity constraints*, that limits some possible combinations of abducibles. More formally:

Definition 14 (Integrity constraint). *A (deterministic) integrity constraint IC is a formula of the form:*

$$: -\text{Body}$$

where Body *is a conjunction of logical literals (logical atoms or their negations).*

Definition 15 (Abductive logic program). *An abductive logic program is a triple* $(P, \mathcal{IC}, A)$ *where* P *is a normal logic program,* $\mathcal{IC}$ *is a set of integrity constraints, and* A *is a set of ground atoms called abducibles that do not appear in the head of any grounding of* P*.*

The goal of abduction is to find *abductive explanations*:

Definition 16 (Abductive explanation). *Given an abductive logic program* $(P, \mathcal{IC}, A)$ *and a conjunction of ground atoms* q *(the query), the goal of abduction is to find a set of atoms* $\Delta \subseteq A$ *called abductive explanation such that* $P \cup \Delta \models q$ *and, for every integrity constraint of the form* $: -\text{Body}_i$*,* $P \cup \Delta \not\models \exists Body_i$*, where* $\models$ *is interpreted as truth in the WFM of the program.*

To manage negation, we require that $P \cup \Delta$ has a two-valued WFM for every Δ. Negation is then defined under the WFM, and $\models$ is well-defined (true or false for any $P \cup \Delta$ and query q). Furthermore, we consider false the abducible facts not present in the abductive explanation.

To clarify the concepts, consider the following simple example:

```
1 flood:- water.
2 water:- broken_pipe.
3 water:- rain.
4
5 :- broken_pipe, rain.
```

where both `broken_pipe` and `rain` are abducibles. The last line represents an integrity constraint imposing that the atoms `broken_pipe` and `rain` cannot be true at the same time. The query `flood` has two abductive explanations, $\Delta_1 = \{$`broken_pipe`$\}$ and $\Delta_2 = \{$`rain`$\}$. The explanation $\Delta_3 = \{$`broken_pipe`,`rain`$\}$ is forbidden by the constraint.

Chapter 6

Syntax and Semantics for Probabilistic Logic Programs

Probabilistic Logic Programming (PLP) [59, 119, 136] extends Logic Programming by allowing uncertainty on the data. Several Probabilistic Logic Programming Languages have been proposed during the years, with different expressive power. Some of them are PRISM [146], CP-Logic [168], Stochastic Logic Programs [116], Independent Choice Logic [128], Probabilistic Horn Abduction [126], ProbLog [60], and LPAD [169]. Here, and in the rest of the thesis, we consider ProbLog and LPAD: Section 6.1 introduces their syntax while Section 6.2 reviews their semantics for both programs with and without function symbols.

6.1 ProbLog and LPADs

Here, we review the Logic Programs with Annotated Disjunctions (LPAD) and ProbLog syntax, starting from the former, since it was proposed first. An LPAD [169] extends a logic program by allowing a finite set of *annotated disjunctive clauses* of the form:

$$h_1 : \Pi_1; h_2 : \Pi_2; \ldots; h_n : \Pi_n \leftarrow \; b_1, \ldots, b_m.$$

where h_i are logical atoms, b_i are logical literals, and $\Pi_i \in [0, 1]$. The Π_is represent the probability of the i-th head. The values of the Π_is should sum

to 1 (and clearly must not exceed this value). If it is not the case, an extra atom that does not appear in the body of any clause is inserted in the head with associated probability $1 - \sum_i \Pi_i$. If $n = 1$ (there is only one head) the clause is non-disjunctive. We can read the previous clause as follows: if, for a grounding of the clause, $b_1, \ldots, b_m$ is true, h_1 is selected with probability Π_1, h_2 with probability Π_2, and so on. For example, the following program models a toss of a fair coin that can land heads of tails with equal probability:

```
1 toss_fair_coin.
2 head:0.5;tails:0.5:- toss_fair_coin.
```

We can then ask for the probability that the coin lands head or tails. The previous program can be rewritten using a non-disjunctive clause as follows:

```
1 toss_fair_coin.
2 head:0.5:- toss_fair_coin.
```

In this case, if we want to compute the probability of tails, we can ask for the probability that the coin does not land heads.

Similarly, ProbLog allows the definition of probabilistic facts with the syntax

$$\Pi :: f.$$

where Π is the probability associated to ground instantiations of the atom f. The example introduced above can be expressed with the ProbLog syntax as:

```
1 0.5::lands_head.
2 toss_fair_coin.
3 head:- toss_fair_coin, lands_head.
```

We can translate a ProbLog program into an LPAD and vice versa. For example, an LPAD can be translated into a ProbLog program [60] by converting each clause with n heads and m variables (denoted with X)

$$h_1 : \Pi_1; h_2 : \Pi_2; \ldots; h_n : \Pi_n \leftarrow B.$$

into a set of $n - 1$ ProbLog probabilistic facts for auxiliary predicates f_i with

arity m, and n clauses, in this way:

$$\pi_1 :: f_1(X).$$
$$\ldots$$
$$\pi_{n-1} :: f_{n-1}(X).$$
$$h_1 \leftarrow \ B, f_1(X).$$
$$h_2 \leftarrow \ B, \sim f_1(X), f_2(X).$$
$$\ldots$$
$$h_n \leftarrow \ B, \sim f_1(X), \ldots \sim f_{n-1}(X).$$

where

$$\pi_i = \frac{\Pi_i}{\prod_{j=1}^{i-1}(1 - \pi_j)}.$$

Consider now a more complicated example:

Example 1 (Chess)**.** *The following LPAD models the possible results of a chess match:*

```
1 player(alice).
2 player(bob).
3 
4 stronger_opponent:0.7.
5 
6 win:0.3;loss:0.5;draw:0.2:-
7     player(X),
8     stronger_opponent.
```

It states that a player `X` *has 0.3 probability to win, 0.5 to loss, and 0.2 to draw a match when he/she plays against a stronger opponent, a situation that happens with probability 0.7. Both* `alice` *and* `bob` *are players. This LPAD can be translated into a ProbLog program as follows:*

```
1 player(alice).
2 player(bob).
3 
4 0.7::stronger_opponent.
5 
```

```
6  0.3::win_fact(X).
7  0.7142::loss_fact(X).
8
9  win:- player(X), stronger_opponent,
10     win_fact(X).
11 loss:- player(X), stronger_opponent,
12     \+ win_fact(X), loss_fact(X).
13 draw:- player(X), stronger_opponent,
14     \+ win_fact(X), \+ loss_fact(X).
```

In both examples, there is a singleton variable `X`*, i.e., a variable that appears only once in a clause. We decided to keep it to better clarity, since we can refer to a substitution θ involving this variable as $\theta = \{X/term\}$.*

6.2 Distribution Semantics

The distribution semantics [148] provides a meaning to probabilistic logic programs. Here, we consider ProbLog programs without function symbols. We now introduce a series of definitions that can be easily extended to LPADs.

Definition 17 (Atomic choice)**.** *An atomic choice indicates whether a grounding $f\theta$ of a probabilistic fact $\Pi :: f$ is selected or not. It is represented with the triple (f, θ, k) where $k \in \{0, 1\}$. If $k = 1$, the grounding is selected, otherwise, it is not.*

A set of atomic choices is consistent if it does not contain two atomic choices $(f, \theta, 0)$ and $(f, \theta, 1)$ for the same probabilistic fact f and substitution θ. That is, only one alternative is selected for a probabilistic fact.

Definition 18 (Composite choice)**.** *A composite choice κ is a consistent set of atomic choices whose probability can be computed as*

$$P(\kappa) = \prod_{(f_i,\theta,1)\in\kappa} \Pi_i \cdot \prod_{(f_i,\theta,0)\in k} (1 - \Pi_i).$$

All the probabilistic facts are considered independent. This may seem a restriction but, in practice, it does not limits the expressivity [136].

Definition 19 (Selection). *A selection contains one atomic choice for every grounding of every probabilistic fact. It is also called total composite choice, and it is usually indicated with σ.*

A selection identifies a logic program called *world* obtained by including the rules and the probabilistic facts corresponding to every atomic choice with $k = 1$. The probability of a world corresponds to the probability of the selection that identifies it. We consider programs without function symbols, so the set of worlds W is finite and the probabilities of all the worlds sum to 1, i.e.:

$$\sum_{w \in W} P(w) = 1.$$

Furthermore, we consider only programs (worlds) with a two-valued well-founded model. We call these programs *sound*. In this way, a *query* (a conjunction of ground atoms) can be only true or false in a world. Finally, the probability of a query q can be computed as follows:

$$P(q) = \underbrace{\sum_{w} P(q, w)}_{\text{Marginalization}} = \underbrace{\sum_{w} P(q \mid w) \cdot P(w)}_{\text{Product rule}} = \sum_{w \models q} P(w)$$

because $P(q \mid w) = 1$ if $w \models q$ according to the WFS, 0 otherwise.

All the three clauses of the ProbLog version of the program shown in Example 1 have two groundings: for the first clause, call it C_1, $C_1\theta_1$ with $\theta_1 = \{X/\texttt{alice}\}$ and $C_1\theta_2$ with $\theta_2 = \{X/\texttt{bob}\}$. Similarly, for the second and the third clause. The whole program has 5 ground probabilistic facts (`stronger_opponent`, `win_fact(alice)`, `win_fact(bob)`, `loss_fact(alice)`, `loss_fact(bob)`) and so $2^5 = 32$ possible worlds. For ease of computation, consider a restricted version of the ProbLog program of Example 1, where the probabilistic fact `loss_fact/1` and the two clauses `loss/0` and `draw/0` have been removed. Consider the query `win`. There are now three probabilistic facts involved, namely `win_fact(alice)`, `win_fact(bob)`, and `stronger_opponent`. They are reported in Table 6.1, where `w_f(a)`, `w_f(b)`, and `s_o` stand respectively for `win_fact(alice)`, `win_fact(bob)`, and `stronger_opponent`. The query is true in three of them (#4, #6, and #8, highlighted in grey), and the probability of the query `win` can be computed as

#	w_f(a)	w_f(b)	s_o	Probability
1	F	F	F	$(1-0.3)\cdot(1-0.3)\cdot(1-0.7)=0.147$
2	F	F	T	$(1-0.3)\cdot(1-0.3)\cdot 0.7=0.343$
3	F	T	F	$(1-0.3)\cdot 0.3\cdot(1-0.7)=0.063$
4	F	T	T	$(1-0.3)\cdot 0.3\cdot 0.7=0.147$
5	T	F	F	$0.3\cdot(1-0.3)\cdot(1-0.7)=0.063$
6	T	F	T	$0.3\cdot(1-0.3)\cdot 0.7=0.147$
7	T	T	F	$0.3\cdot 0.3\cdot(1-0.7)=0.027$
8	T	T	T	$0.3\cdot 0.3\cdot 0.7=0.063$

Table 6.1: Worlds for a restricted version of the ProbLog program of Example 1, where the probabilistic fact `loss_fact/1` and the two clauses `loss/0` and `draw/0` have been removed. Highlighted rows represent the worlds where the query `win` is true, together with their probability.

$0.147+0.147+0.063=0.357$.

6.2.1 Semantics for Programs with Function Symbols

If a (probabilistic) logic program has at least one constant and a function symbol, both the Herbrand universe and the set of possible groundings of probabilistic facts are denumerable, and the grounding of the program is infinite. The set of worlds is uncountable [21], and the probability of each world is 0, since it is computed as an infinite product of values less than 1. So, the previously discussed semantics is not appropriate. To see this, consider the following example, inspired by [169]:

Example 2 (Game of dice). *A player repeatedly throws a three-sided die. Each round is identified by an index (* `0`, `s(0)`, `s(s(0))`, *...). The game stops when the outcome is three. We can encode this scenario with a ProbLog program as follows:*

```
1 1/3::one(X).
2 1/2::two(X).
3
4 on(0,1) :- one(0).
5 on(0,2) :- \+ one(0), two(0).
6 on(0,3) :- \+ one(0), \+ two(0).
7 on(s(X),1) :- on(X,_), \+ on(X,3), one(s(X)).
```

```
 8 on(s(X),2) :- on(X,_), \+ on(X,3), \+ one(s(X))
      , two(s(X)).
 9 on(s(X),3) :- on(X,_), \+ on(X,3), \+ one(s(X))
      , \+ two(s(X)).
10
11 at_least_once_1 :- on(_,1).
12 never_1 :- \+ at_least_once_1.
```

We may be interested in computing the probability that the die lands at least one time on face 1, and so we need to consider the query `at_least_once_1`*. Similarly, to get the probability that the die never lands on face 1, the query to consider is* `never_1`*. In both cases, we need to manage an infinite number of rounds. Note that the probability of* `two(X)` *has been set to* $1/2$*, since, in this way, the probability of landing on face two and three is given by* $2/3-1/2 = 1/3$*, and* $2/3 - 1/2 = 1/3$ *(the die is fair, all the faces have the same probability).*

To give a semantics to Example 2, we need to introduce some more definitions. We denote with W_P the set of all worlds for a probabilistic logic program P. The set of worlds ω_κ *compatible* with a composite choice κ is $\omega_\kappa = \{w_\sigma \in W_P \mid \kappa \subseteq \sigma\}$. For programs with function symbols, ω_κ may be uncountable and $\sum_{w \in \omega_k} P(w)$ could be undefined since $P(w) = 0$. Let us rename $P(\kappa)$ in $\mu(\kappa)$. The set of worlds ω_K *compatible* with a set of composite choices K is defined as $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$. Two sets K_1 and K_2 of composite choices are equivalent if $\omega_{K_1} = \omega_{K_2}$ (they correspond to the same set of worlds). Two composite choices are *incompatible* if their union is inconsistent. The following definition is crucial in giving a semantics to programs with function symbols.

Definition 20 (Pairwise incompatible set of composite choices)**.** *A set* K *of composite choices is pairwise incompatible if* $\forall \kappa_1, \kappa_2 \in K,\ \kappa_1 \neq \kappa_2 \implies \kappa_1$ *and* κ_2 *are incompatible. That is, every pair of composite choices is incompatible.*

For probabilistic logic programs (with and without function symbols) obtaining pairwise incompatible sets of composite choices is of key importance. The probability of a pairwise incompatible set of composite choices K for programs without function symbols is defined as $P(K) = \sum_{\kappa \in K} P(\kappa)$ and can be

easily computed. In case the program has function symbols, if K is countable then $P(K)$ is well-defined. As before, let us rename $P(K)$ in $\mu(K)$.

If $f\theta$ is an instantiated fact and κ is a composite choice that does not contain the atomic choices $(f, \theta, 0)$ and $(f, \theta, 1)$, the *split* of κ on $f\theta$ is defined as the set of composite choices $S_{\kappa,f\theta} = \{\kappa \cup \{(f, \theta, 0)\}, \kappa \cup \{(f, \theta, 1)\}\}$. With this operation, $\omega_\kappa = \omega_{S_{\kappa,f\theta}}$. That is, the set of worlds identified by κ and $S_{\kappa,f\theta}$ are the same. Furthermore, $S_{\kappa,f\theta}$ is pairwise incompatible.

Through the technique of *splitting* it is possible to assign a probability to a general set K of composite choices. It works as follows: if $f\theta$ is an instantiated fact and κ is a composite choice that does not contain an atomic choice $(f, \theta, 0)$ and $(f, \theta, 1)$, the *split* of κ on $f\theta$ is the set of composite choices $\{\kappa \cup \{(f, \theta, 0)\}, \kappa \cup \{(f, \theta, 1)\}\}$. By iteratively applying it, we can obtain an equivalent mutually incompatible set of composite choices [127], as stated by these two following theorems.

Theorem 1 (Existence of a pairwise incompatible set of composite choices [127])**.** *Given a finite set K of composite choices, there exists a finite set K' of pairwise incompatible composite choices equivalent to K.*

Theorem 2 (Equivalence of the probability of two equivalent pairwise incompatible finite set of finite composite choices [126])**.** *If K_1 and K_2 are both pairwise incompatible finite sets of finite composite choices and they are equivalent, then $P(K_1) = P(K_2)$.*

Given a finite pairwise incompatible set of composite choices K' equivalent to K it is possible to define a measure μ_P for a probabilistic logic program P as $\mu_P(\omega_K) = \mu(K')$. To see how, we need to introduce more definitions.

Definition 21 (Explanation)**.** *A composite choice κ is an explanation for a query q if, $\forall w \in \omega_\kappa,\ w \models q$.*

Definition 22 (Covering set of composite choices)**.** *A set of composite choices K is covering with respect to a query q if every world in which q is true belongs to ω_K.*

To clarify these two concepts, consider this example.

Example 3 (Pairwise incompatible covering set of explanations for Example 2). *If we denote* `one(X)` *with* f_1 *and* `two(X)` *with* f_2*, the query* `at_least_once_1` *in Example 2 has the pairwise incompatible covering set of explanations*

$$K^+ = \{\kappa_0^+, \kappa_1^+, \ldots\}$$

with

$$\begin{aligned}
\kappa_0^+ &= \{(f_1, \{X/0\}, 1)\}\\
\kappa_1^+ &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), (f_1, \{X/s(0)\}, 1)\}\\
&\ldots\\
\kappa_i^+ &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), \ldots, (f_1, \{X/s^{i-1}(0)\}, 0),\\
&\quad (f_2, \{X/s^{i-1}(0)\}, 1), (f_1, \{X/s^i(0)\}, 1)\}\\
&\ldots
\end{aligned}$$

So K^+ *is countable and infinite. Similarly, the query* `never_1` *has the pairwise incompatible covering set of explanations*

$$K^- = \{\kappa_0^-, \kappa_1^-, \ldots\}$$

with

$$\begin{aligned}
\kappa_0^- &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 0)\}\\
\kappa_1^- &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), (f_1, \{X/s(0)\}, 0),\\
&\quad (f_2, \{X/s(0)\}, 0)\}\\
&\ldots\\
\kappa_i^- &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), \ldots, (f_1, \{X/s^{i-1}(0)\}, 0),\\
&\quad (f_2, \{X/s^{i-1}(0)\}, 1), (f_1, \{X/s^i(0)\}, 0), (f_2, \{X/s^i(0)\}, 0)\}\\
&\ldots
\end{aligned}$$

We call Ω_P the set of worlds identified by countable sets of countable composite choices for a probabilistic logic program P, i.e., $\Omega_P = \{\omega_K \mid K \text{ is a countable set of countable composite choices}\}$. Lemma 2 of [136] proves that Ω_P is a σ-algebra over W_P. We can define a probability measure as

$\mu_P : \Omega_P \rightarrow [0, 1]$.

Given a (possibly infinite) set of composite choices $K = \{\kappa_1, \kappa_2, \dots\}$, consider the sequence $\{K_n \mid n \geq 1\}$ where $K_n = \{\kappa_1, \dots, \kappa_n\}$. K_n is an increasing sequence, and so $\lim_{n\rightarrow\infty} K_n$ exists and is $\bigcup_{n=1}^{\infty} K_n = K$ [47]. We can construct a sequence $\{K_n' \mid n \geq 1\}$ in this way: $K_1' = \{\kappa_1\}$, and K_n' is given by the union of K_{n-1}' with the splitting of each element of K_{n-1}' with κ_n. The obtained set K_n' is pairwise incompatible and equivalent to K_n [136]. For infinite composite choices, $\mu(\kappa) = 0$. However, we can compute $\mu(K_n')$ for each K_n', and the limit $\lim_{n\rightarrow\infty} \mu(K_n')$ exists (limit of the measure of countable union of countable composite choices, Lemma 3 from [136]).

Finally, we provide a definition of a probability space of a program with the following theorem:

Theorem 3 (Probability space of a program, Theorem 8 from [136])**.** *Given a set of composite choices $K = \{\kappa_1, \kappa_2, \dots\}$ and a pairwise incompatible set of composite choices K_n' equivalent to $\{\kappa_1, \dots, \kappa_n\}$, the triple $\langle W_P, \Omega_P, \mu_P\rangle$ with*

$$\mu_P(\omega_K) = \lim_{n\rightarrow\infty} \mu(K_n')$$

is a probability space.

For a probabilistic logic program P and a ground atom q, we define the function $Q : W_P \rightarrow \{0, 1\}$ as

$$Q(w) = \begin{cases} 1 & \text{if } w \models q \\ 0 & \text{otherwise} \end{cases} \quad (6.1)$$

If K is a covering and countable set of explanations with respect to q, Equation 6.1 represents a random variable, since $\{w \mid w \in W_P \wedge w \models q\} = \omega_K \in \Omega_P$. We indicate $P(Q = 1)$ with $P(q)$, and we say that q is well-defined according to the distribution semantics. If the probabilities of all ground atoms in all instances of a probabilistic logic program P are well-defined, then P is also well-defined. In [135, 136] the author proved that any query to a sound ProbLog program can be assigned a probability so that the program is well-defined.

To see how to compute the probability of a query in the case of an infinite number of explanations, consider the following example:

Example 4 (Probability of queries for Example 2)**.** *From Example 3, the explanations in K^+ are pairwise incompatible so the probability of the query* `at_least_once_1` *can be computed as:*

$$
\begin{aligned}
P(at_least_once_1) = {} & \mu(\{(f_1, \{X/0\}, 1)\}) + \\
& + \mu(\{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), \\
& (f_1, \{X/s(0)\}, 1)\}) + \ldots
\end{aligned}
$$

By substituting the values, we get:

$$
\begin{aligned}
P(at_least_once_1) &= \frac{1}{3} + \frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right) + \frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right)^2 + \ldots \\
&= \frac{1}{3} + \frac{1}{3} \cdot \left(\frac{1}{3}\right) + \frac{1}{3} \cdot \left(\frac{1}{3}\right)^2 + \ldots \\
&= \frac{1}{3} \cdot \frac{1}{1 - \frac{1}{3}} = \frac{1}{3} \cdot \frac{3}{2} = \frac{1}{2}
\end{aligned}
$$

since the sum represents a geometric series and $\sum_{n=0}^{\infty} k \cdot q^n = k \cdot \frac{1}{1-q}$. For the query `never_1`*, the explanations in K^- are pairwise incompatible, so its probability can be computed as*

$$
\begin{aligned}
P(never_1) = {} & \frac{2}{3} \cdot \frac{1}{2} + \frac{2}{3} \cdot \frac{1}{2} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right) + \\
& \frac{2}{3} \cdot \frac{1}{2} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right)^2 + \ldots \\
= {} & \frac{1}{3} + \frac{1}{3} \cdot \left(\frac{1}{3}\right) + \frac{1}{3} \cdot \left(\frac{1}{3}\right)^2 + \ldots \\
= {} & \frac{1}{3} \cdot \frac{1}{1 - \frac{1}{3}} = \frac{1}{3} \cdot \frac{3}{2} = \frac{1}{2}.
\end{aligned}
$$

As expected, $P(never_1) = 1 - P(at_least_once_1)$.

6.3 Conclusions

In this section, we introduced the syntax of LPADs and ProbLog programs, together with their semantics. If a program does not have function symbols,

its grounding is finite and so, according to the Distribution Semantics, the probability of a query can be computed by considering the possible worlds. In case of function symbols, the semantics is not straightforward, and the probability of a query can be computed by considering pairwise incompatible covering sets of explanations.

In the next section, we review the main techniques used to compute the probability of a query, as well as some algorithms for approximating its value.

Chapter 7

Inference

Inference is the task of computing the probability distribution of the truth values of a query. There are two types of inference: *exact* inference (Section 7.1) that computes exact values, and *approximate* inference (Section 7.2), that provides an estimation of the values. In the context of Probabilistic Logic Programming, the first type usually rewrites the program into a compact representation where performing inference is tractable (Section 7.1.1), while the second type is often based on sampling (Section 7.2.1). In this chapter, we compare these two inference techniques and describe how they are used to query probabilistic logic programs. The concepts described here are not a novel contribution of this thesis, except for the approaches reported in Section 7.2.1 that were introduced in [19, 23].

7.1 Exact Inference

The goal of exact inference is to compute the probability of a query (conjunction of ground atoms) in an exact way, as we have already done in Section 6.2. However, an exact computation is not always feasible. To compute the probability of a restriction of Example 1, we have considered all the possible combinations of n probabilistic facts (3), which are 2^n (8, see Table 6.1). Clearly, an exponentially increasing number of combinations is not tractable, or it is only for smaller domains. Moreover, queries can also consider *evidence* (represented as a conjunction of ground atoms). In this case, the value of some variables is observed and thus fixed.

Exact inference is #P-complete [97] since it inherits the cost of inference in the underlying graphical model. To manage this, some techniques that go under the name of *knowledge compilation* [56] have been introduced. Knowledge compilation (KC) [56] consists in converting a propositional theory into a target language that allows answering queries in polynomial time. Note that the computational complexity is simply shifted from one task to another and remains the same in the worst case. Propositional theories are usually one of these two representations:

- Conjunctive Normal Form (CNF): a conjunction of one or more clauses where each clause is a disjunction of literals. For example, $(a \lor b \lor \neg c) \land (\neg d \lor e)$ is a formula in CNF.
- Disjunctive Normal Form (DNF): a disjunction of one or more clauses where each clause is a conjunction of literals. For example, $(a \land b \land \neg c) \lor (\neg d \land e)$ is a formula in DNF.

There is another popular representation called Negation Normal Form (NFF) where sentences are represented as rooted acyclic graphs [56], but we do not review it here.

Consider this simple example, taken from [35], that models the morphological characteristics of an island.

Example 5. *The island of Stromboli contains one of the three volcanoes that are active in Italy. It is located at the intersection of two geological faults, one in the southwest-northeast direction, the other in the east-west direction. This LPAD models the possibility that an eruption or an earthquake occurs at Stromboli.*

```
1 eruption:0.6;earthquake:0.3 :- sudden_er,
     fault_rupture(X).
2 sudden_er:0.7.
3 fault_rupture(southwest_northeast).
4 fault_rupture(east_west).
```

If there is a sudden energy release (`sudden_er`*) under the island and there is a fault rupture (*`fault_rupture(X)`*), there can be an eruption of the volcano on the island with probability 0.6 or an earthquake in the area with probability 0.3.*

The energy release occurs with probability 0.7, and we are sure that ruptures occur along both faults.

An example of composite choice that is also an explanation for the query `eruption` of Example 5 is $\kappa_1 = \{(C_1, \{X/southwest_northeast\}, 1), (C_2, \emptyset, 1)\}$, where C_1 and C_2 are the clauses at line respectively 1 and 2. A covering set of explanations for the query `eruption` is:

$$\kappa_1 = \{(C_1, \{X/southwest_northeast\}, 1), (C_2, \emptyset, 1)\};$$
$$\kappa_2 = \{(C_1, \{X/east_west\}, 1), (C_2, \emptyset, 1)\}.$$

Given a covering set of explanations for a query, we can convert it into a DNF form following these three steps:

- Replace every atomic choice (C_i, θ_j, k) with the equation $X_{ij} = k$.
- Replace an explanation with the conjunction of the equations represented by its atomic choices.
- Represent the set of explanations as the disjunction of all the formulas for a single explanations.

If worlds are considered as specification of truth values for each equation $X_{ij} = k$, the formula evaluates to true exactly on the worlds where the query is true [127]. The probability of a query can be computed as the probability that this formula takes value 1, since random variables are independent and have a known distribution (discrete). Consider again Example 5: if we associate variable X_{11} with $C_1\{X/southwest_northeast\}$, variable X_{12} with $C_1\{X/east_west\}$, and variable X_{21} with $C_2\emptyset$, the query is true if the following Boolean formula is true:

$$f(\mathbf{X}) = (X_{21} = 1 \land X_{11} = 1) \lor (X_{21} = 1 \land X_{12} = 1). \tag{7.1}$$

The DNF representation of a program does not guarantee that the obtained formulas are mutually exclusive, so we cannot compute the probability as a summation of products of probabilities. This problem is called *disjoint sum* and its complexity is in the #P-complete class [163]. One way to tackle this

complexity is by representing a DNF formula with Decision Diagrams, that we introduce next.

7.1.1 Decision Diagrams

Decision Diagrams (DDs) are structures used to compactly represent propositional logic formulas. There are several types of DDs: here, we focus on Multi-valued Decision Diagrams and Binary Decision Diagrams.

A Multi-valued Decision Diagram (MDD) represents a function $f(X)$ that can take Boolean values on a set of multi-valued variables X through a rooted graph, where each level is associated to a different variable. Each node n has as many children as the number of possible values of the variable associated to the level of the node n can take. Leaves, however, can only store 0 or 1. For example, Figure 7.1a shows an example of MDD corresponding to Formula 7.1. The branches of the MDD are mutually exclusive, so we can use a dynamic programming algorithm to compute the probability of a query given the MDD representation of the program [144].

Most software packages that operate on Decision Diagrams are restricted to Binary Decision Diagrams (BDDs), where all the variables are Boolean. Differently from MDDs, a node n in a BDD has only two children: the 0-child, and the 1-child. To represent an MDD through a BDD, we adopt the encoding proposed in [144]: for a multi-valued variable X_{ij} corresponding to a ground clause $C_i\theta_j$ having n_i possible values we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$, and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ with the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ with the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. See also the approach to translate an LPAD into a ProbLog program described in Section 6.1. For example, Figure 7.1b depicts a BDD equivalent to the MDD of Figure 7.1a, where the edge going to the 0-child is drawn with a dashed line while the edge going to the 1-child with a straight line. The obtained BBD can still be used for computing the probability of queries by associating a parameter π_{ik} with each Boolean variable X_{ijk} representing $P(X_{ijk} = 1)$. The parameters can be computed starting from those of multi-valued variables in this way: $\pi_{i1} = \Pi_{i1}, \ldots, \pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}$, up to $k = n_i - 1$.

Some BDD software libraries introduce a third type of edge, the *comple-*

mented edge to a 0-child, with the meaning that the function represented by the child must be complemented, following this rule: if the leaf value is 1, and in the path starting from the root and reaching this leaf we passed through an odd number of complemented edges, the value 0 must be considered (instead of 1). With this third edge, only the 1-leaf is needed. Figure 7.1c shows an example of BDD with complemented edges encoding the formula $(X_0 \wedge X_1) \vee (\overline{X_0} \wedge \overline{X_2})$. One of the libraries that adopt this approach is CUDD[1], that we extensively use in this thesis to implement several algorithms discussed in the next sections.

With the term BDD we refer to Reduced Ordered Binary Decision Diagrams (as often happens in literature), i.e., BDDs where the order of the variables is fixed. A BDD is *ordered* if the variables encountered along the paths always respect a given total order $X_1 \prec \cdots \prec X_n$. A BDD is ordered if the following conditions are satisfied:

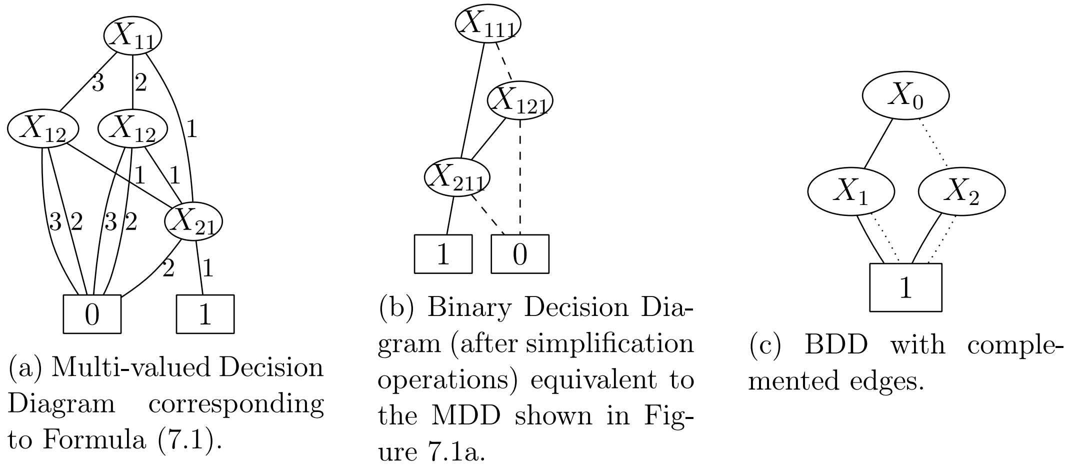
- Two distinct nodes on a path cannot be associated to the same variable.
- Two distinct nodes cannot have the same 0 and 1 children.
- The 0 and 1 children are different for all the variables.

The order of the variables in a BDD is crucial for its succinctness: two different variable orderings may generate BDDs with a different number of nodes (since the same variable can be represented with several nodes of the same level). Finding the order of variables leading to the smallest diagram is NP-complete [37]. However, reordering a BDD by swapping adjacent variables can be performed polynomially in its size [84].

7.1.2 Systems to Perform Exact Probabilistic Logical Inference

One of the first systems that used BDDs to perform inference in probabilistic logic programs was ProbLog1 [60]. The authors introduced the function PROB, reported in Algorithm 1, for computing the probability that a Boolean function, represented by a BDD, takes value 1. The function recursively traverses the BDD staring from the root and until a terminal is found. Then, going

[1] `https://github.com/ivmai/cudd`



(a) Multi-valued Decision Diagram corresponding to Formula (7.1).

(b) Binary Decision Diagram (after simplification operations) equivalent to the MDD shown in Figure 7.1a.

(c) BDD with complemented edges.

Figure 7.1: Decision Diagrams.

back to the root, the probability of every node is computed as the sum of the contributions of the 0 and 1 children multiplied respectively by the probability and one minus the probability associated to the current node. To avoid computing the same values multiple times (there can be multiple paths that share the same sub-path), intermediate results are stored in a table.

Algorithm 1 Function PROB: computation of the probability of a BDD.

```
1: function PROB(node, TableProb)
2:     if node is a terminal then
3:         return 1
4:     else
5:         if TableProb(node.pointer) ≠ null then
6:             return TableProb(node)
7:         else
8:             p0 ←PROB(child0(node), TableProb)
9:             p1 ←PROB(child1(node), TableProb)
10:             if child0(node).comp then
11:                 p0 ← (1 − p0)
12:             end if
13:             Let π be the probability of being true of var(node)
14:             Res ← p1 · π + p0 · (1 − π)
15:             Add node.pointer → Res to TableProb
16:             return Res
17:         end if
18:     end if
19: end function
```

Similarly to ProbLog1, the PITA reasoner [139] performs inference in LPADs by converting them into BDDs. Moreover, it uses *tabling* [159], a logic programming technique that saves already computed answers for a goal, to avoid recomputing them when they are needed. These answers can also be combined using *answer subsumption* [159]. ProbLog2 [64] is a newer version of

ProbLog1, where BDDs have been replaced by Deterministic Decomposable Negation Normal Forms (d-DNNF). In this system, the program is converted into a weighted Boolean formula (where literals have an associated weight) and inference is performed through Weighted Model Counting (WMC) [45].

7.2 Approximate Inference

While exact inference is always desirable, it is feasible only for small domains, since the explanations for a query may be in a large number. In other cases, the probability distribution defined by the program is too complex to evaluate exactly (for example, when there is a mixture of continuous and discrete random variables), so approximate methods are of interest. Furthermore, a program may have an infinite number of groundings: in this case, approximate inference is necessary. For example, consider a one-dimensional random walk where a particle starts at position $X \neq 0$ and, at each step, it can move one unit left or right with the same probability. The walk stops when the particle reaches 0. Here, 0 is reached with probability 1 [78], but there is an infinite number of paths with non-zero probability that reach the terminal state [90]. In a program modelling this scenario, exact inference tries to find all the explanations, but it will loop forever since there is an infinite number of them.

Several solutions have been proposed during the years, based on different techniques: *iterative deepening* [60], which limits the depth of the SLD tree for the proof of a query, *k-best* [95], that considers only a fixed number of proofs to provide a lower bound of the probability, and Monte Carlo methods [60, 134, 166], based on sampling. Here we focus on the last type.

A general algorithm that adopts sampling repeatedly executes these three steps for a fixed number of times or until convergence (often reached when the difference of two consecutive computed values is below a certain threshold):

- Samples a world by sampling ground probabilistic facts.
- Checks if the query is true in the world.
- Computes the probability of the query as the number of successes divided by the number of samples taken so far.

Approximate algorithms for probabilistic logic programs (LPADs) are implemented in the MCINTYRE module [134] of `cplint` [2]. The program to sample is converted into a modified one and queries are asked in this modified version. A disjunctive clause

$$h_1 : \Pi_1; h_2 : \Pi_2; \ldots ; h_n : \Pi_n \leftarrow \; b_1, \ldots , b_m$$

is transformed into a set of clauses $\{MC(C, 1), \ldots , MC(C, m)\}$ where

$$\begin{aligned} MC(C,1) =& h_1 :- \; b_1, \ldots , b_m, \\ & sample_head(PL, i, VC, NH), NH = 1. \\ \ldots & \\ MC(C,m) =& h_m :- \; b_1, \ldots , b_n, \\ & sample_head(PL, i, VC, NH), NH = m. \end{aligned}$$

where i is the index of the clause, VC is a list containing each variable appearing in the clause, and PL is the list containing the probabilities of every head ($\{\Pi_1, \ldots , \Pi_n\}$). That is, for each head, a new clause with the previously described structure is generated. If the Π_is do not sum to 1, the clause for the auxiliary extra atom (see Section 6.1) is omitted. The MCINTYRE predicate `sample_head/4` samples a head index according to the probabilities. This operation is performed at the end of the body: in this way, all the variables in VC have been already grounded (supposing that the program is range restricted). If the sampled value is the same as the head index, the derivation succeeds; otherwise it fails. The collected samples are stored in the dynamic Prolog database using the predicate `assertz/1` provided by SWI-Prolog [171]. Finally, the truth of a query can be tested against the resulting program, which is equivalent to taking a sample of the query. For example, if we consider this following simple program:

```
1 f(a).
2 f(b).
3 p::0.7.
4 a:0.6;b:0.3 :- f(X), p.
```

the clause at line 4 is transformed into two clauses:

```
1 a:- f(X), p,
2     sample_head([0.6,0.3,0.1],[X],NH), NH = 1.
3 b:- f(X), p,
4     sample_head([0.6,0.3,0.1],[X],NH), NH = 2.
```

We now focus on several possible algorithms used to compute the probability of a query given evidence ($P(q \mid e)$) by sampling. *Rejection sampling* [97] is one of the simplest algorithms based on Monte Carlo methods. It works in two steps: 1) it queries (samples) the evidence. If it is successful, 2) it queries the goal in the same sample; otherwise the sample is discarded. The pseudocode for this procedure is shown in Algorithm 2. Despite its simplicity, rejection sampling has a huge drawback: if the probability of the evidence $P(e)$ is very low, a lot of samples are discarded, making the algorithm very inefficient. For example, if $P(e) = 10^{-4}$ and the number of samples we take is 10^5, the expected number of not rejected samples is 10. In general, to obtain at least N not rejected samples, we need to generate $N/P(e)$ samples from the probability distribution. There are some alternative algorithms that better handle this scenario, such as *likelihood weighting* and *Markov Chain Monte Carlo*. In the next section, we focus on the latter.

Algorithm 2 Function REJECTIONSAMPLING: rejection sampling algorithm.

1: **function** REJECTIONSAMPLING($P, query, evidence, samples$)
2: Input: Probabilistic logic program P, query $query$, evidence $evidence$, number of samples $samples$
3: Output: $P(query \mid evidence)$
4: $succ \leftarrow 0$
5: $n \leftarrow 1$
6: **while** $n \leq samples$ **do**
7: Call $evidence$
8: **if** $evidence$ succeeds **then**
9: Call $query$
10: **if** $query$ succeeds **then**
11: $succ \leftarrow succ + 1$
12: **end if**
13: $n \leftarrow n + 1$
14: **end if**
15: **end while**
16: **return** $succ/samples$
17: **end function**

7.2.1 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) methods generate samples from the posterior distribution when directly sampling from it is infeasible, due to its com-

plexity. The main idea is to iteratively construct a Markov Chain in which direct sampling is easy: the higher the number of samples is, the better the approximation of the true posterior will be. Initial samples are often discarded since they do not properly represent the real distribution. This operation is called *burnin*. In the limit of infinite samples, MCMC can get arbitrarily close to the true posterior. Two of the most used MCMC algorithms are Metropolis Hastings sampling and Gibbs sampling, that we now introduce.

Metropolis Hastings

In Metropolis Hastings sampling, a Markov chain is built by generating successor samples starting from an initial sample. We consider the algorithm for inference in PLP presented in [118] and implemented in `cplint` [2, 23]. After applying the same program transformation presented for MCINTYRE, it iteratively repeats these steps:

- To build an initial sample, it samples random choices so that the evidence is true.
- To build a successor sample it removes a fixed number (called *lag*) of sampled probabilistic choices.
- It queries again the evidence by sampling starting from not deleted choices.
- If the evidence succeeds, the query is asked by sampling. It is accepted with probability $min\{1, N_0/N_1\}$, where N_0 is the number of choices sampled in the previous sample and N_1 is the number of choices sampled in the current sample.
- If the query succeeds in the last accepted sample, the number of successes is increased by one.

The final probability is computed as the ratio between the number of successful samples and the number of total samples. Algorithm 3 implements the procedure. In detail, the function MH returns the probability of the query given the evidence. The function RESAMPLE deletes *lag* number of choices from the sample random choices. In [118], the value of lag is always 1. The function

Algorithm 3 Function MH: Metropolis-Hastings MCMC algorithm.

```
1: function MH(query, evidence, lag, samples)
2:     MHCycle(query, evidence, lag)
3:     return MHCycle(query, evidence, samples)
4: end function
5: function MHCycle(query, evidence, samples)
6:     trueSamples ← 0
7:     sample ← InitialSample(evidence)
8:     Call query
9:     if query succeeds then                              ▷ Sample atomic choices
10:        trueSamples ← trueSamples + 1
11:    end if
12:    let previousSampled be the current number of choices sampled
13:    Save a copy of the current samples C      ▷ Save the samples for probabilistic clauses
14:    n ← 0
15:    while n < samples do
16:        n ← n + 1
17:        Resample(lag)
18:        Call evidence
19:        if evidence succeeds then
20:            Call query
21:            if query succeeds then
22:                trueSamples ← trueSamples + 1
23:            end if
24:            let currentSampled be the current number of choices sampled
25:            if min(1, currentSampled/previousSampled) > RandomValue(0, 1) then
26:                previousSampled ← currentSampled
27:                Delete the copy of the previous samples C
28:                Save a copy of the current samples C
29:            else
30:                Erase all samples
31:                Restore samples copy C
32:            end if
33:        else
34:            Erase all samples
35:            Restore samples copy C
36:        end if
37:    end while
38:    Erase all samples
39:    Delete the copy of the previous samples C
40:    return trueSamples/samples
41: end function
42: procedure Resample(lag)
43:    for n ← 1 to lag do
44:        Delete a sample sample
45:        newSample ← Sample(sample)
46:        Assert newSample
47:    end for
48: end procedure
```

InitialSample builds the initial sample using a meta-interpreter that starts with the goal and randomizes the order in which clauses are used during the resolution: this is achieved by first collecting all the clauses that match a subgoal and then trying them in random order. This operation is needed to make the initial sample unbiased. Finally, the goal is queried using regular sampling.

Gibbs Sampling

The main idea behind Gibbs sampling is the following: when direct sampling from a joint distribution is not feasible, we can sample each variable independently while considering all the other as observed [69]. If we have n variables $X_1, \ldots, X_n$, we can set their initial value to $x_1^{(0)}, \ldots, x_n^{(0)}$ by sampling from a prior distribution (for example). At each iteration, or until convergence, we take a sample $x_m^{(t)} \sim P(x_m \mid x_1^{t-1}, x_2^{t-1}, \ldots, x_{m-1}^{t-1}, x_{m+1}^{t-1}, \ldots, x_n^{t-1})$. That is, all the variables except for the current one are considered observed. It is also possible to group two or more variables together and sample from their joint distribution conditionally on all the others: in this case, the algorithm goes under the name of *blocked* Gibbs sampling. Algorithm 4 shows the implementation of Gibbs sampling in `cplint`. It goes as follows: the list of sampled random choices is stored in the Prolog dynamic database using `assertz/1`, as for Metropolis Hastings. The function GibbsCycle performs the main loop: the function SampleCycle executes a type of rejection sampling to query the evidence by iteratively querying it until the value true is obtained. When a successful sample for probabilistic clauses is obtained, we remove *block* random choices from the list of saved random choices using the function CheckSamples: it checks if there are some rules not sampled in the list of removed random choices and, if so, a value is sampled and stored in memory. This step is needed since we need to assign a value to X_m even if it is not directly involved in the derivation of the query. By sampling the clauses in this way we follow the Gibbs technique. Finally, as for rejection sampling and Metropolis Hastings, the probability is computed as the number of successes over the number of total samples.

Comparison of rejection sampling, Metropolis Hastings sampling, and Gibbs sampling

We compared the performance of the three previously described algorithms using the predicates `mc_rejection_sample/5`, `mc_mh_sample/5`, and `mc_gibbs_sample/5` provided by the MCINTYRE module [134]. They perform the type of sampling appearing in their names. The following results are presented in [23]. All the algorithms are implemented in Prolog and tested in

Algorithm 4 Function GIBBS: Gibbs MCMC algorithm.

```
1: function GIBBS(query, evidence, mixing, samples, block)
2:     GIBBSCYCLE(query, evidence, mixing, block)
3:     return GIBBSCYCLE(query, evidence, samples, block)
4: end function
5: function GIBBSCYCLE(query, evidence, samples, block)
6:     succ ← 0
7:     for n ← 1 to samples do
8:         Save a copy of samples C
9:         SAMPLECYCLE(evidence)
10:        Delete the copy of samples C
11:        listOfRemovedSamples ← REMOVESAMPLES(block)
12:        Call query                    ▷ New samples are asserted at the bottom of the list
13:        if query succeeds then
14:            succ ← succ + 1
15:        end if
16:        CHECKSAMPLES(listOfRemovedSamples)
17:    end for
18:    return Succ/samples
19: end function
20: procedure SAMPLECYCLE(evidence)
21:    while true do
22:        Call evidence
23:        if evidence succeeds then
24:            return
25:        end if
26:        Erase all samples
27:        Restore samples copy C
28:    end while
29: end procedure
30: function REMOVESAMPLES(block)
31:    sampleList ← []
32:    for b ← 1 to block do
33:        retract sample S = (rule, substitution, value)    ▷ Samples are retracted from the top of the list
34:        Add (rule, substitution) to sampleList
35:    end for
36:    return sampleList
37: end function
38: procedure CHECKSAMPLES(listOfRemovedSamples)
39:    for all (rule, substitution) ∈ listOfRemovedSamples do
40:        if (rule, substitution) was not sampled then
41:            Sample a value for (rule, substitution) and record it with assert
42:        end if
43:    end for
44: end procedure
```

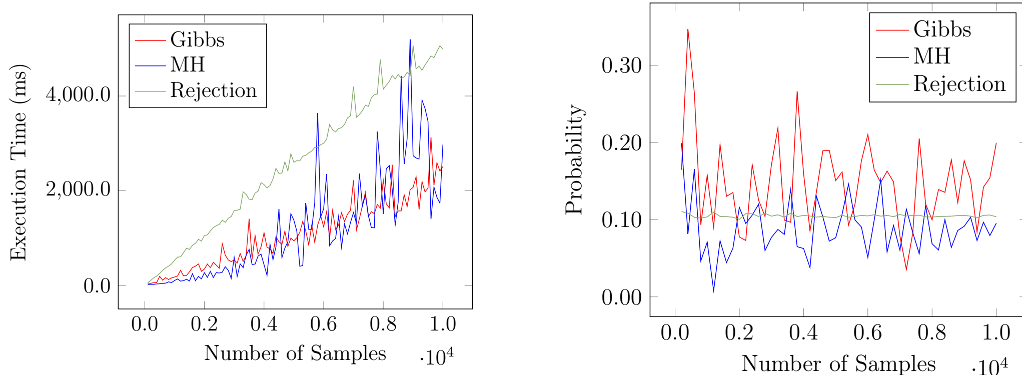
SWI-Prolog [171] version 8.1.7. The experiments were conducted on a cluster[2] with Intel® Xeon® E5-2630v3 running at 2.40 GHz. Execution times are computed using the built-in SWI-Prolog predicate `statistics/2`[3] with the keyword `walltime`. For both Metropolis Hastings sampling and Gibbs sampling, we set the burnin to 100 (number of deleted samples). For Gibbs sampling, we set the *block* value to 1. The probability of the evidence is computed with the `cplint` predicate `mc_sample/3` that samples the query a fixed

[2] http://www.fe.infn.it/coka/doku.php?id=start

[3] https://www.swi-prolog.org/pldoc/man?predicate=statistics/2

number of times and returns the ratio between number of successes and total samples. We set the number of samples to 10^6. Reported results are averages of ten runs.

Tests were conducted on four different types of programs. The first program encodes a random arithmetic function[4] (arithm): the goal is to predict the returned value given one or two couples input-output. This program has an infinite number of explanations, so exact inference cannot be applied. In these tests, the probability of the evidence was set to 0.05. Figure 7.2 shows the results: both Metropolis Hastings sampling and Gibbs sampling have comparable execution times, but the latter has a slower convergence rate.



(a) Relation between execution time and number of samples.

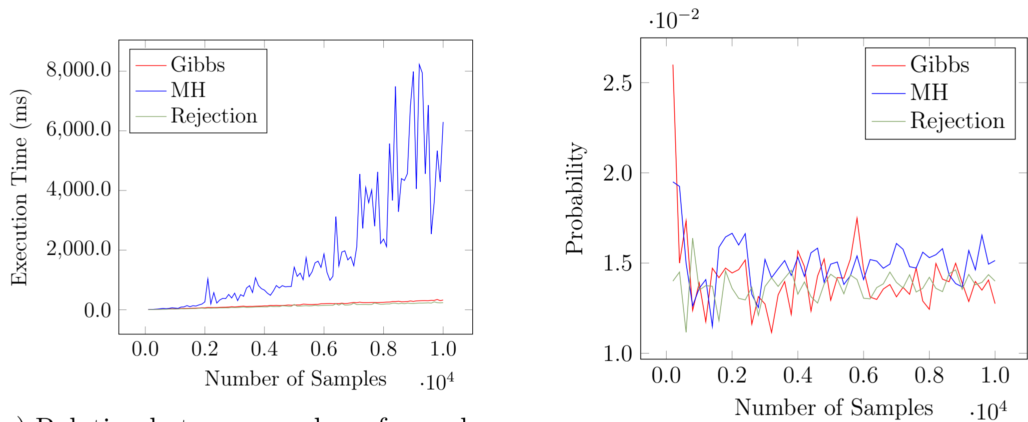
(b) Relation between computed probability and number of samples.

Figure 7.2: Results for the arithm experiment.

The second program[5] encodes a Hidden Markov model (HMM) used to model DNA sequences. It has three states, $q1$, $q2$, and *end*, and four output symbols, a, c, g, and t, corresponding to the four nucleotides [49]. The goal is to compute the probability that the model emits a and then c given that the model emitted a in state $q1$. The evidence has probability 0.25. As shown in Figure 7.3, all the three algorithms converge to the same value, but the execution time of MH is orders of magnitude bigger than the other two.

In the third experiment, we considered a Latent Dirichlet Allocation (LDA) model[6] [36]. LDA is a generative probabilistic model used in text analysis: it

[4] `https://cplint.eu/e/arithm.pl`
[5] `https://cplint.eu/e/hmm.pl`
[6] `https://cplint.eu/e/lda.swinb`



(a) Relation between number of samples and execution time.

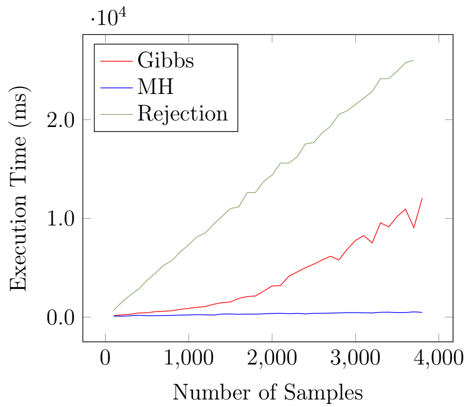
(b) Relation between number of samples and computed probability.

Figure 7.3: Results for the HMM experiment.

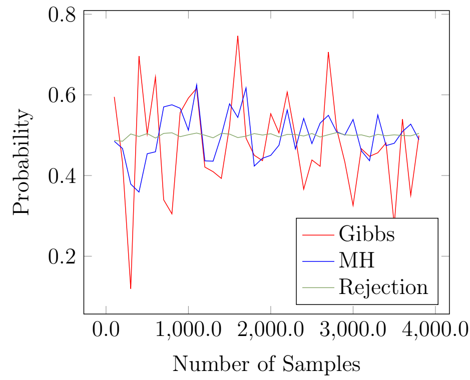
models the distribution of terms and topics in documents with the goal to predict the topic of the considered text. This program contains both continuous and discrete random variables, so it is a *hybrid* program[7]. In this test, we both fixed the number of topics (2), and the number of words considered in a document (10), and we computed the relation among the number of samples, probability, and execution time, given that the first two words of a document are equal (probability 0.01). Figure 7.4 shows that Gibbs sampling is slower than MH, and it requires more samples to compute an accurate probability. In a second test on the same program, we increased the number of words that are equal (evidence), from 2 to 8, while keeping unchanged the number of considered words and topics. Also in this case, MH outperforms the other two algorithms, as reported in Figure 7.5a. The number of words in the plot for Gibbs sampling and rejection sampling is at most 6 since, for higher values, the computation requires more than one hour.

The last program represents a university domain[8] [110] characterized by professors, students, and courses. Each professor is assigned to a course, and each student attends a course. Given that a professor is advisor of some students following a course, we want to know the probability that the professor teaches that course. We fixed the number of students to 10 and considered a

[7]Hybrid programs will be extensively analysed in Chapter 8.
[8]`https://cplint.eu/e/uwcse.pl`

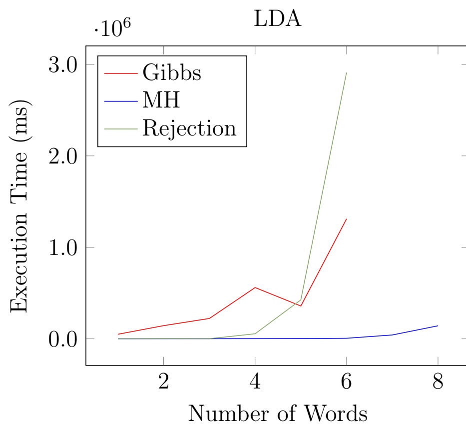


(a) Relation between number of samples and execution time.

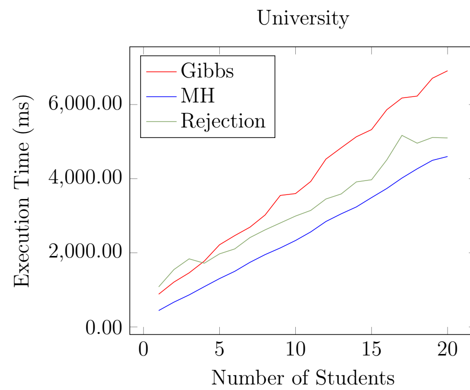


(b) Relation between number of samples and computed probability.

Figure 7.4: Results for the LDA experiment.



(a) Relation between number of words and execution time for the LDA experiment.

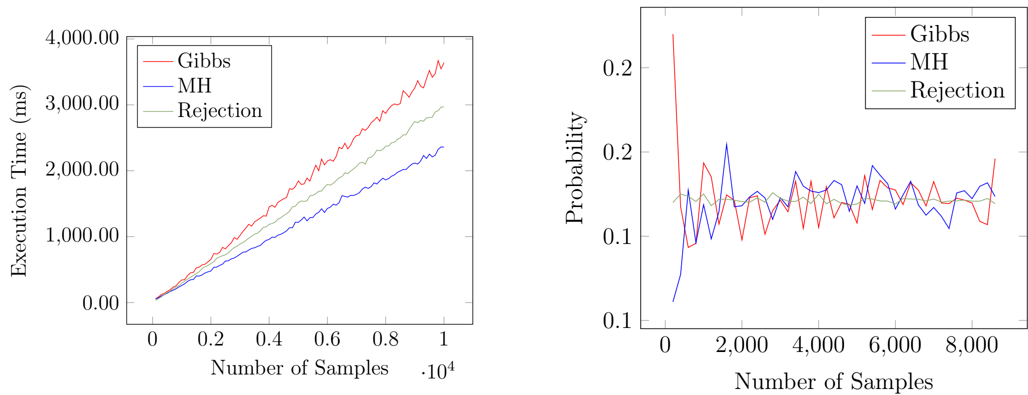


(b) Relation between number of students and execution time for the university experiment.

Figure 7.5: Results for the LDA and university experiments. For both we fixed the number of samples to 10^4.

single course and a single professor. The probability of the evidence is 0.09. As figures 7.5b and 7.6a show, Gibbs sampling is the slowest among the three, but the performance gap is not too critical, also in the case of an increasing number of students.

Overall, in three of the four experiments, MH outperformed the other two algorithms both in terms of accuracy and execution time, except for the HMM experiment, where Gibbs sampling seems the fastest (together with rejection



(a) Relation between number of samples and execution time.

(b) Relation between number of samples and computed probability.

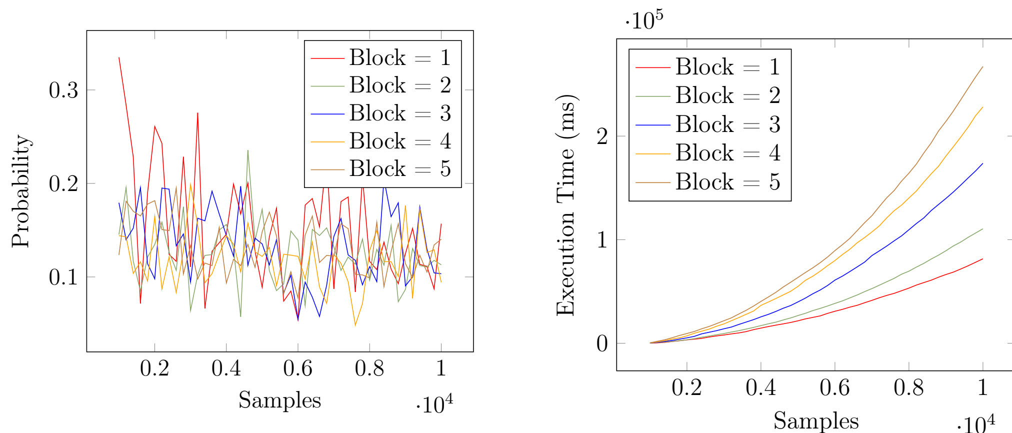
Figure 7.6: Results for the university experiment.

sampling) and the most accurate, while MH overestimates the probability. According to our results, if the evidence has a relatively low probability, Metropolis Hastings sampling is the fastest among the three. If the probability of the evidence increases, Gibbs sampling seems to perform better.

We now focus our attention on the performance of blocked Gibbs sampling, where two or more variables are sampled together. The following results are presented in [19]. We run the Gibbs sampling algorithm on eight different datasets. For each one of them, we plotted three graphs: one to represent the performance in terms of number of samples required to converge, one to track the execution time, and one to control the standard deviation of the samples. The experiments were conducted with the same setting as before. The number of discarded initial samples was set to 100, and the block size ranges from 1 to 5. We tested eight different datasets. The first one is arithm, the same as before, with the probability of evidence still set to 0.05. As Figure 7.7a shows, the probability value computed with block set to 1 oscillates between 0.1 and 0.25, as also confirmed by the instability of the standard deviation shown in Figure 7.9a. This oscillation is still present, but less pronounced, with increasing values of block.

Diabetes[9]: this program models the probability of insurgence of diabetes given that some genetic factors are observed. For example, the diabetes pre-

[9] `https://cplint.eu/e/diabetes.swinb`



(a) Relation between number of samples and computed probability.

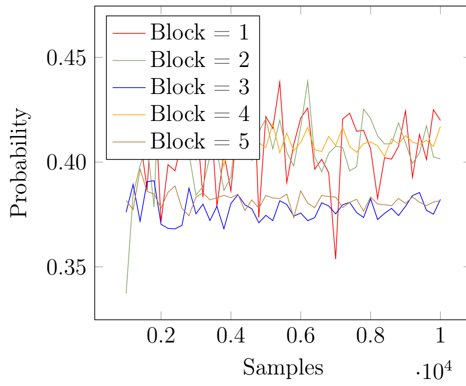
(b) Relation between number of samples and execution time.

Figure 7.7: Results for the arithm experiment.

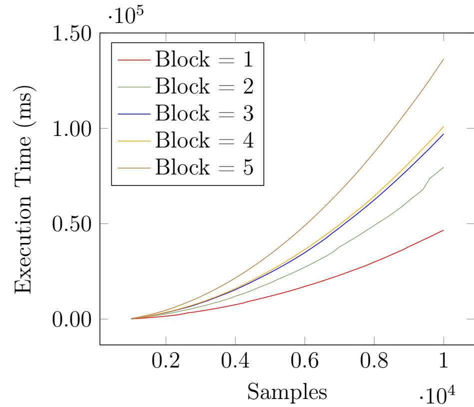
disposition of a person influences the probability of diabetes mellitus of type 2. This is an example of a probabilistic constraint logic program [112], i.e., a program that contains continuous random variables and constraints. In fact, the level of glucose is modelled with two Gaussian distributions, with different means and variances, depending on the fact that the considered person has diabetes or not. A constraint is present to model the level of hemoglobin since it linearly depends on the level of glucose (plus some noise). We observe that the level of hemoglobin is greater than a certain value (probability 0.1417), and we want to compute how the probability that a person has diabetes type 2 varies with and without evidence. For this experiment, smaller sizes of block usually drive to lower accuracy and significant variation of the standard deviation, as reported in Figure 7.8a and Figure 7.9b. With block set to 5, this variation is no longer present, but at the cost of increasing execution time. When the block value is set to 3 or 5, the algorithm seems to underestimate the probability.

Graph[10]: this program encodes a graph following a Barabási Albert preferential attachment model. Given a graph with some initially connected nodes, new nodes are added to the graph and the probability that these new nodes are connected to others is proportional to the degree (number of incident edges) of a node. The graph was generated with the function

[10] `https://cplint.eu/e/barabasiGraph.pl`

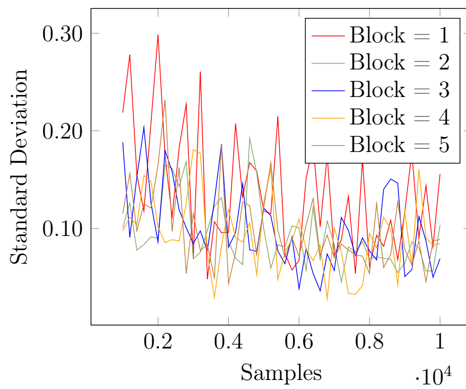


(a) Relation between number of samples and computed probability.

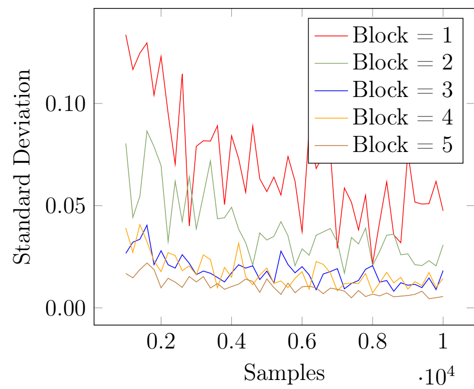


(b) Relation between number of samples and execution time.

Figure 7.8: Results for the diabetes experiment.



(a) Results for the arithm experiment.

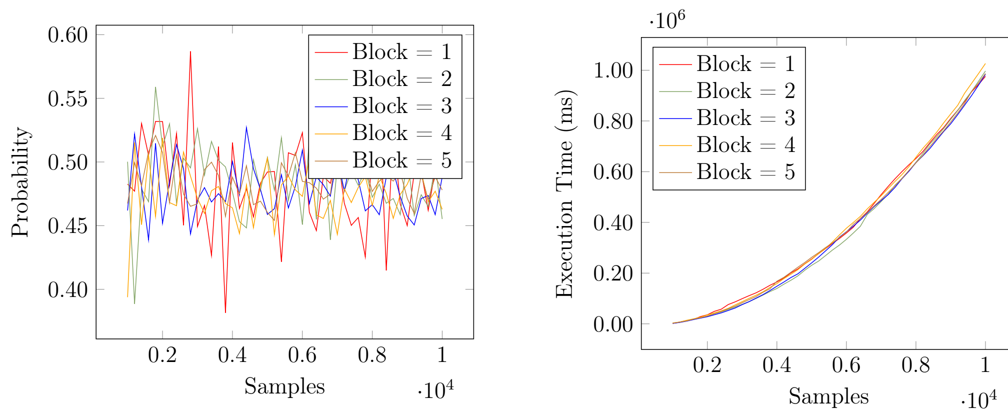


(b) Results for the diabetes experiment.

Figure 7.9: Standard deviation for the arithm and diabetes experiments.

`barabasi_albert_graph(40,10)` provided by the library `networkx` [75] available for Python. We associated a probability of 0.1 to all the edges. The query consists in computing the probability that two nodes are connected given that a portion of the path has been observed. In our tests, the probability of the evidence was 0.42. For this example, the probability of the query, as well as the one of the evidence, can be computed using exact inference. However, when the number of nodes and edges increases, an exact computation becomes infeasible. As for the previous program, with smaller values of block we obtain unstable probability values: for example, with block set to 1, there is a gap of 0.2 between two measurements, see Figure 7.10a. The execution times for

all the block values are comparable, while the standard deviation decreases as the block size increases (Figure 7.12a). The computed probability values with block set to 3, 4, and 5, are, in practice, the same.



(a) Relation between number of samples and computed probability.

(b) Relation between number of samples and execution time.

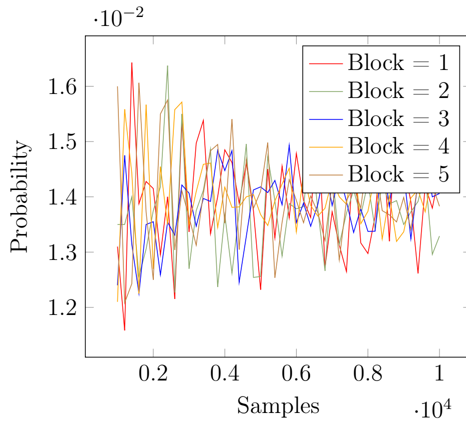
Figure 7.10: Results for the graph experiment.

Hidden Markov Model, with the same setting as before: as Figure 7.11 shows, all the block values perform well, and the computed probabilities do not vary too much, even with a small number of samples. Only in the case of block set to 1 the algorithm requires some time to stabilize, as also described by the standard deviation plot (Figure 7.12b). The execution times for block set to 4 and 5 are almost identical.

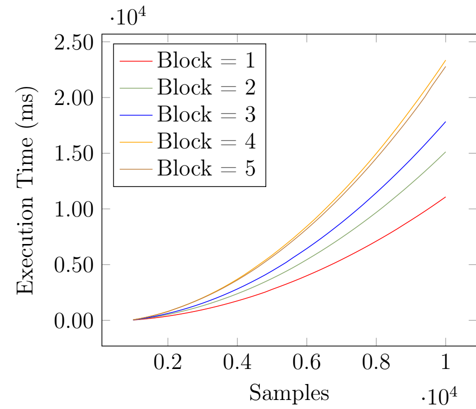
Latent Dirichlet Allocation, with the same setting as before (10 words and 2 topics): we want to compute the probability that the considered document associates the first topic to the first word given that we observed the type of the word (probability of evidence 0.10). Results in figures 7.4 and 7.15a show that even with 10^4 samples the probability does not stabilize, regardless the block size, as for the standard deviation.

NBalls[11]: this program models an urn containing n balls, where n is a random variable, characterized by colour, material, and size, with known distributions. As a query, we want to compute the probability that we pick a wooden ball at the first draw given that we observe its colour (black, probability 0.38). In this case, the probability stabilizes only after 10^4 samples and only

[11] `https://cplint.eu/e/nballs.pl`

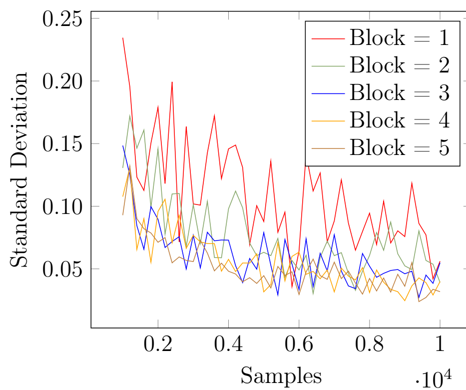


(a) Relation between number of samples and computed probability.

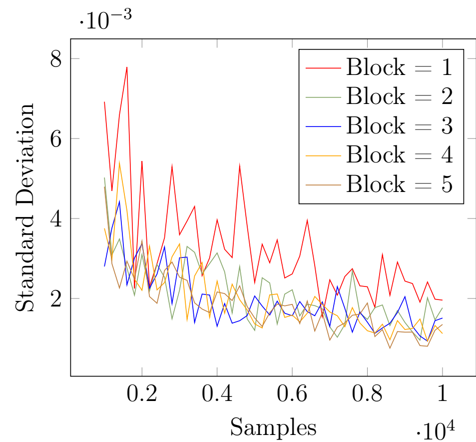


(b) Relation between number of samples and execution time.

Figure 7.11: Results for the HMM experiment.



(a) Results for the graph experiment.



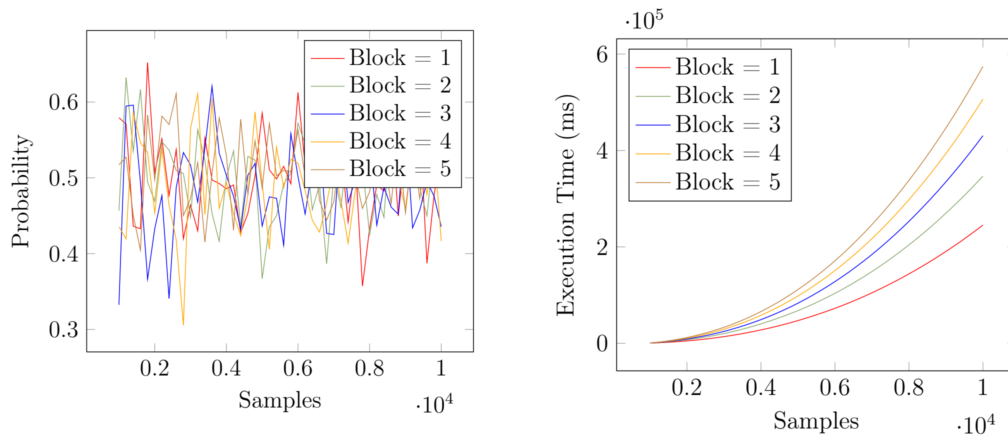
(b) Results for the HMM experiment.

Figure 7.12: Standard deviation for the graph and HMM experiments.

for bigger values of block, as shown in Figure 7.14a. Block sizes set to 1 and 3 seem to be the most unstable. However, for all block values, the standard deviation decreases as the number of samples increases (Figure 7.15b).

Prefix parser[12]: this program models a prefix parser for context free grammars [151], i.e., it computes the probability that a string is a prefix of another string generated by the grammar. In the program, we considered a simple grammar composed of only two letters, `a`, and `b`, and observed that the first emitted is `a` (probability 0.5). The goal is to compute the probability of the

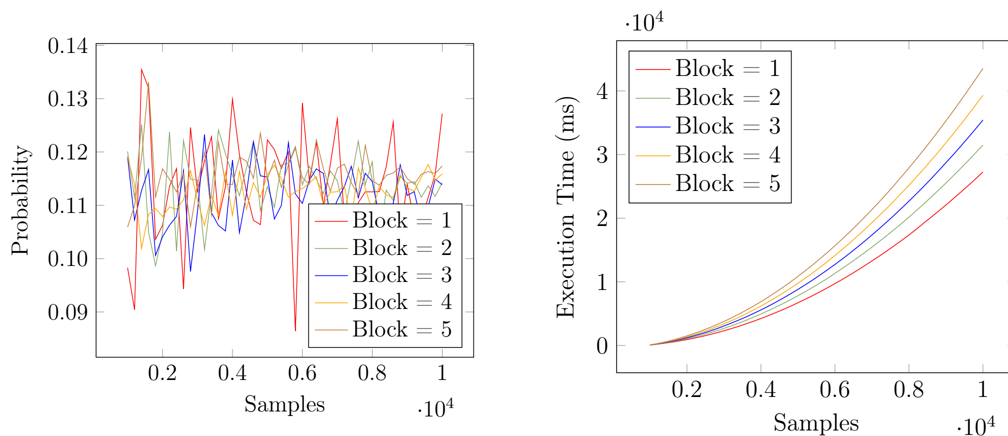
[12] `https://cplint.eu/e/prefix.pl`



(a) Relation between number of samples and computed probability.

(b) Relation between number of samples and execution time.

Figure 7.13: Results fot the LDA experiment.



(a) Relation between number of samples and computed probability.

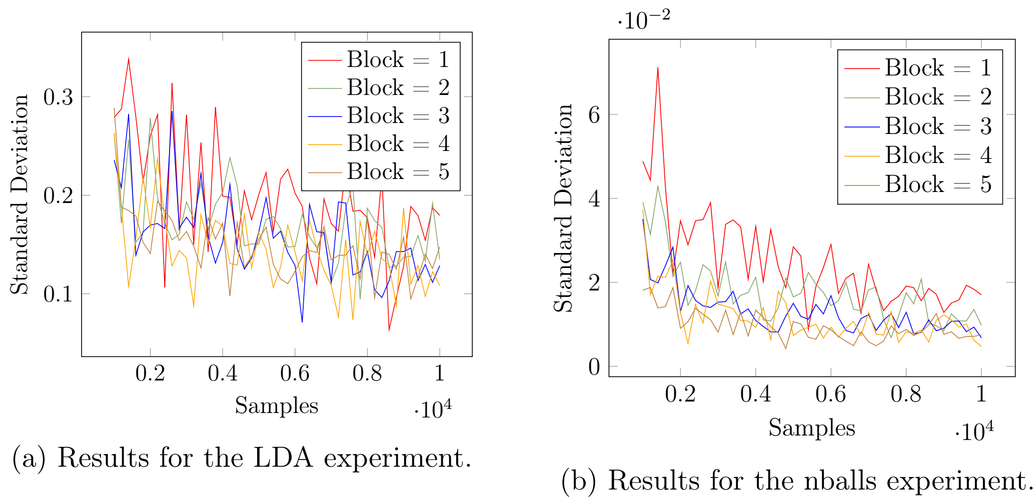
(b) Relation between number of samples and execution time.

Figure 7.14: Results for the nballs experiment.

string `[a,b,a]`. As reported in figures 7.16 and 7.18a, the conditional probability is small, but all the five values of block seem to perform equally well. The execution times of these experiments are the greatest among the eight.

Stochastic Logic Program[13]: this program defines a probability distribution over possible sentences. A characteristic of Stochastic Logic Programs (SLP) is that no stochastic memoization is performed, meaning that repeated choices are independent. Furthermore, the probabilities of rules with the same

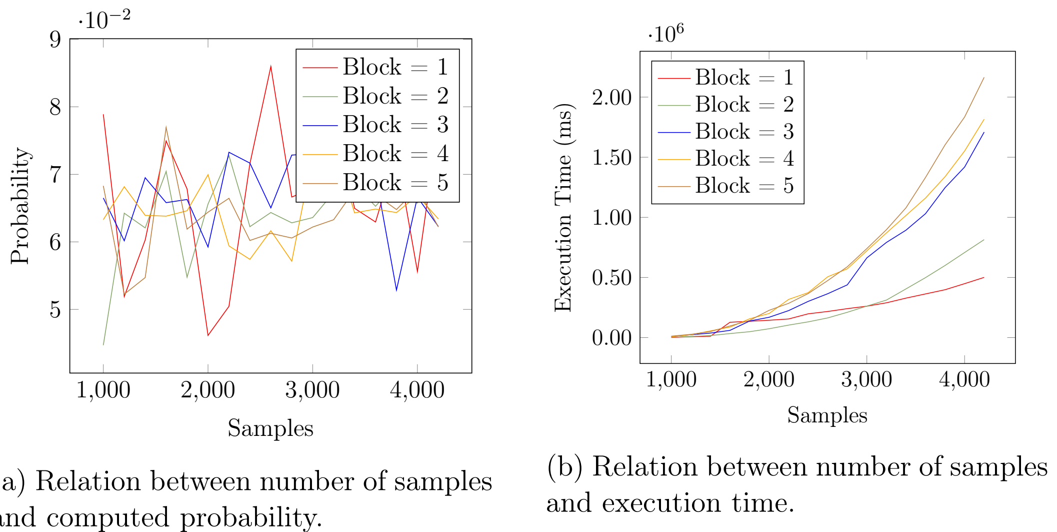
[13] `https://cplint.eu/e/slp_pdcg.pl`



(a) Results for the LDA experiment.

(b) Results for the nballs experiment.

Figure 7.15: Standard deviation for the LDA and nballs experiments.



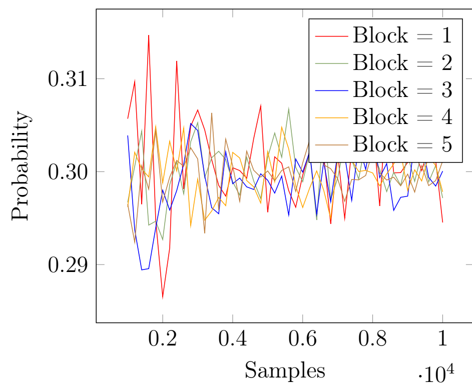
(a) Relation between number of samples and computed probability.

(b) Relation between number of samples and execution time.

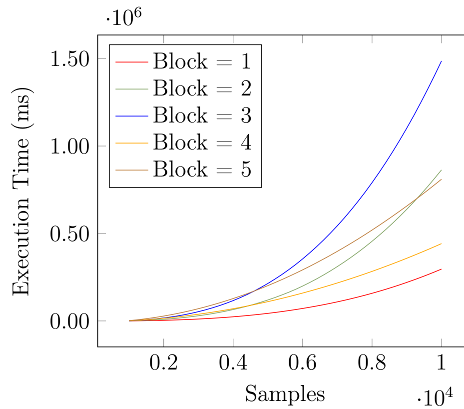
Figure 7.16: Results for the prefix parser experiment.

head sum to 1 and these rules are mutually exclusive. For the query, we are interested in computing the probability that three particular words are sampled given that we observe the first one (probability 0.006). As reported in Figure 7.17a, the probability stabilizes for all the block values after a few thousands of samples. As expected, the standard deviation decreases as the number of samples increases (Figure 7.18b). For this experiment, the block size set to 3 requires the greatest execution time.

Overall, when the block value increases, both the probability and the stan-

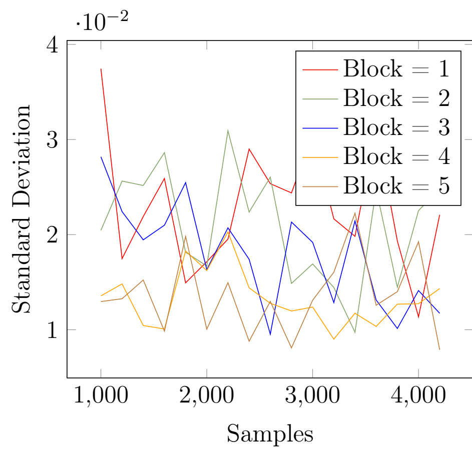


(a) Relation between number of samples and computed probability.

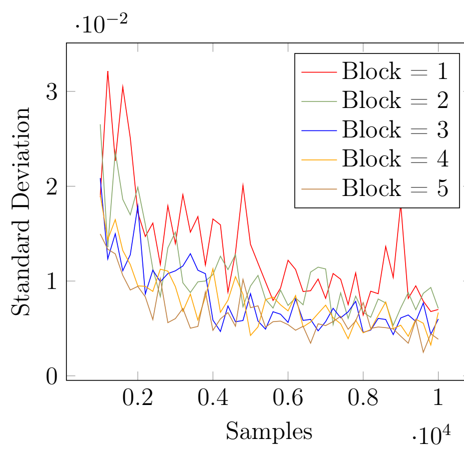


(b) Relation between number of samples and execution time.

Figure 7.17: Results for the stochastic logic program experiment.



(a) Results for the prefix parser experiment.



(b) Results for the stochastic logic program experiment.

Figure 7.18: Standard deviation for the prefix parser and stochastic logic program experiments.

dard deviation seem to require fewer samples to stabilize. However, an increment of the block size is often linked to an increment of the execution time.

7.3 Conclusions

In this chapter, we analysed the problem of inference in probabilistic logic programs. Exact inference is feasible only when the size of the program is

small, due to the necessity to compute all explanations for the query. When the size increases, approximate techniques must be adopted. In this perspective, we tested the implementation of three algorithms: rejection sampling, Metropolis Hastings sampling, and Gibbs sampling (the latter two belonging to the Markov Chain Monte Carlo family). Our results show that, in general, Metropolis Hastings is the fastest among the three when the evidence has a low probability. However, when the probability of the evidence increases, Gibbs sampling seems to have good performance. For Gibbs sampling, we also studied its behavior in cases where two or more variables are sampled together (*blocked* sampling). In this case, according to our results, the probability requires more samples to stabilize.

Part III

Extensions of Probabilistic Logic Programming

Chapter 8

Hybrid Programs

Traditional probabilistic logic programs, as discussed in Chapter 6, manage only discrete random variables. However, several real-world scenarios require introducing continuous random variables to represent measurements such as speed or temperature. In this chapter, we review several languages for *hybrid* probabilistic logic programs and provide a new formal semantics (Section 8.2) together with the necessary syntactic requirements (Section 8.2.2) to preserve its well-definedness. The content of this chapter, except for the analysis of the already existing languages, is novel, and was introduced in [15, 21].

8.1 Hybrid Probabilistic Logic Programs

With the term *hybrid* programs we denote a class of (probabilistic logic) programs where both discrete and continuous random variables coexist. Several languages have been proposed, together with different semantics. One of the first languages to allow continuous random variables was Hybrid ProbLog [70]. *Continuous* probabilistic facts adopt the syntax:

$$(X, \phi) :: f$$

where X is a logical variable called *continuous random variable* that appears in the atom f, and ϕ is an atom that specifies the type of the distribution (in [70], only Gaussian distributions were considered). For example, the continuous

probabilistic fact:

$$(X, gaussian(0, 1)) :: f(X)$$

states that X follows a Gaussian distribution with mean 0 and variance 1. A Hybrid ProbLog program is composed of a set of definite rules R and a set of probabilistic facts $F = F^d \cup F^c$, where F^d is the set of discrete facts (as in ProbLog), and F^c is the set of continuous facts. The language provides some predicates to represent constraints on continuous random variables. For example, given a continuous variable X and two numeric constants v_1 and v_2, the predicates $above(X, v_1)$ and $below(X, v_1)$ succeed if the value of X is respectively greater than or smaller than v_1. Similarly, $ininterval(V, n_1, n_2)$ succeeds if $v_1 \leq X \leq v_2$. The semantics of these programs allows only a finite set of continuous probabilistic facts and therefore no function symbols. The set of continuous random variables $X = \{X_1, \ldots, X_n\}$ is defined by the set of atoms for probabilistic facts $F^c = \{f_1, \ldots, f_n\}$, were each f_i is ground except for variable X_i, and each X_i has an associated probability density $p_i(X_i)$. An assignment $x = \{x_1, \ldots, x_n\}$ to X defines a substitution $\theta_x = \{X_1/x_1, \ldots, X_n/x_n\}$ and a set of ground facts $F\theta_x$. A world $w_{\sigma,x}$ is defined as $w_{\sigma,x} = R \cup \{f\theta \mid (f, \theta, 1) \in \sigma\} \cup F\theta_x$, where σ is a selection for discrete facts, and x is an assignment to continuous variables. All continuous variables are independent, so the probability of an assignment $p(x)$ can be computed as $p(x) = \prod_i p_i(x_i)$. $p(x)$ is a joint probability density over X, so $P(\sigma)$ and $p(x)$ define a joint probability density over the worlds:

$$p(w_{\sigma,x}) = p(x) \prod_{(f_i,\theta,1)\in\sigma} \Pi_i \prod_{(f_i,\theta,0)\in\sigma} 1 - \Pi_i$$

where Π_i is the probability associated with the discrete fact f_i. The probability of a ground atom q, that must not be an atom of a continuous probabilistic fact, can be defined as in the Distribution Semantics for discrete programs:

$$P(q) = \sum_{\sigma \in S_P} \int_{x \in \mathbb{R}^n : w_{\sigma,x} \models q} p(w_{\sigma,x})\ dx$$

where S_P is the set of all selections over discrete probabilistic facts. If the set $\{(\sigma, x) \mid \sigma \in S_P, x \in \mathbb{R}^n : w_{\sigma,x} \models q\}$ is measurable, $P(q)$ is well-defined.

For each instance σ, the set $\{x \mid x \in \mathbb{R}^n : w_{\sigma,x} \models q\}$ can be considered as an n-dimensional interval of the form $I = \bigtimes_{i=1}^{n}[a_i, b_i]$ on $\mathbb{R}^n$. $-\infty$ and $+\infty$ are allowed for a_i and b_i respectively. With these conventions, the probability that $X \in I$ can be computed as

$$P(X \in I) = \int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} p(x)\ dx.$$

Two of the limitations of this language are: 1) function symbols are not allowed, and 2) constraints between random variables are not permitted.

Distributional Clauses (DC) [73] is another language that can be used to express continuous random variables[1]. A Distributional Clause is composed of a set of definite clauses of the form $h \sim D \leftarrow b_1, \dots, b_n$, where D is a possibly non ground term used to specify the probability distribution of the term h. Each ground instance $C\theta$ of a DC C defines a random variable $h\theta$ with distribution $D\theta$ if the body $(b_1, \dots, b_n)\theta$ is true. Similarly to Hybrid ProbLog, DC provides some predicates, called *rel_preds*, to compare the outcome of a random variable with constants or outcomes of other random variables. A DC program is composed of a set of definite clauses and a set of Distributional Clauses. The set of definite clauses, together with the set of true ground atoms for the predicates in *rel_preds* for each random variable in the program, defines a world. The semantics of DC programs is based on a stochastic extension of the T_p operator (see Section 5.2), called ST_P. The authors define a function called ReadTable to evaluate probabilistic facts and store sampled values for random variables. When applied to a probabilistic fact, it returns the truth value of the fact according to the values of the random variables passed as arguments: if the value is present in the table, it is returned, otherwise, a new value is computed. More formally, the ST_P operator applied to a program P and a set of ground facts I is defined as [73]:

$$\begin{aligned} ST_P(I) = \{h \mid h \leftarrow b_1 \dots, b_n \in \mathit{ground}(P) \land \forall b_i : (b_i \in I \lor \\ (b_i = \mathit{rel}(t_1, t_2) \land \\ (t_j =\simeq h \Rightarrow (h \sim D) \in I) \land \textsc{ReadTable}(b_i) = \mathit{true}))\} \end{aligned}$$

[1]We use Distributional Clauses to indicate both the name of the language and the type of clauses composing a program in this language.

where $rel \in \{=, <, >, \leq, \geq\}$. DC programs do not allow negations in the body of rules, and several constraints involving rounding of variables must be met to preserve the validity of the semantics.

The authors of [121] extended the previously defined ST_P operator applied to a program P (still limited to clauses without negations in the body) to a set I of ground facts and equalities of the form $t = v$:

$$
\begin{aligned}
ST_P(I) = \{h = v \mid\ & h \sim D \leftarrow b_1, \ldots, b_n \in ground(P) \wedge \forall b_i : \\
& (b_i \in I \vee b_i = rel(t_1, t_2) \wedge t_1 = v_1 \in I \wedge t_2 = v_2 \in I \wedge \\
& rel(v_1, v_2) \wedge v \text{ is sampled from } D)\} \cup \\
& \{h \mid h \leftarrow b_1, \ldots, b_n \in ground(P) \wedge h \neq (r \sim D) \wedge \forall b_i : \\
& (b_i \in I \vee b_i = rel(t_1, t_2) \wedge t_1 = v_1 \in I \wedge \\
& t_2 = v_2 \in I \wedge rel(v_1, v_2))\}.
\end{aligned}
$$

For each DC clause, when the body is true in I, the ST_P operator samples a value v from the specified distribution for the random variable in the head h and adds the relation $h = v$ to the interpretation. In case of deterministic clauses, new ground atoms are added to the interpretation if the body is true. We can obtain a model of an instance of a DC program by computing the least fixpoint of the ST_P operator. Moreover, the ST_P operator is stochastic, so it represents a sampling process that defines a probability distribution over truth values of a query.

The DC semantics without limitations on the type of literals in the body of clauses is at the heart of HAL-ProbLog [174]. A clause in this language is represented as $D :: t \leftarrow l_1, \ldots, l_n$. Given a grounding substitution θ, $t\theta$ represents a continuous random variable following the distribution D if the body $(l_1, \ldots, l_n)\theta$ is true. To avoid the definition of random variables that follows two or more distributions, clauses with the same head must have mutually exclusive bodies. Also for this language, there exist two built-in predicates to handle continuous random variables: $valS(RV, LV)$, that unifies the random variable RV with the logical variable LV representing its value, and $conS(Exp)$, where Exp denotes a constraint imposed on logical variables. For example, $valS(v, V), conS(V > 3)$ imposes that the value of the random variable v must be greater than 3. This semantics also does not allow function

symbols.

Extended PRISM [81] extends the PRISM language [148] to allow Gamma and Gaussian distributions. These can be introduced with the directive set_sw. For example, $set_sw(v, norm(0, 1))$ states that the value of the variable v follows a Gaussian distribution with mean 0 and variance 1. It is also possible to define linear equality constraints over reals. The inference algorithm proposed in [81] symbolically reasons over the imposed constraints exploiting the specified restrictions. The semantics of these programs is based on an extension of the Distribution Semantics with the least model semantics of constraint logic programs [83]. Starting from msw atoms, the probability space is extended to a probability space for the entire program. The sample space of a single continuous random variable is $\mathbb{R}$, and it is extended to the product of sample spaces in case of a set of random variables. The probability space of n continuous random variables is defined as the Borel σ-algebra [47] over $\mathbb{R}^n$, and the Lebesgue measure is used as probability measure. As for DC, negation is not allowed in the body of clauses.

8.1.1 Probabilistic Constraint Logic Programming

We now focus on another proposal to manage continuous random variables and constraints, namely Probabilistic Constraint Logic Programming (PCLP) [112]. A probabilistic constraint logic program is composed of a countable set of random variables X, where each element $X_i \in X$ has an associated range $Range_i$ (discrete, $\mathbb{R}$, or $\mathbb{R}^n$), and a set of rules R that defines the truth value of the atoms in the Herbrand base of the program given the values of the random variables. The sample space of a set X of random variables is defined as $W_X = \bigtimes_{i=1}^{|X|} Range_i$. Each random variable X_i has an associated probability space $(Range_i, \Omega, \mu_i)$. The measure space (W_X, Ω) is the product of the measure spaces $(Range_i, \Omega_i)$ for every random variable i, so it is an infinite-dimensional product measure space [47]. For any finite subset of the set of random variables it is possible to generate a probability space as the product of the involved probability spaces. Starting from these new probability spaces, we can build an infinite dimensional probability space (W_X, Ω_X, μ_X) by extending them (Theorem 6.4.1 from [47]). A constraint ϕ is a function $\phi : W_X \to \{true, false\}$. The *constraint solution space* $CSS(\phi)$ is the subset

of the sample space W_X where the constraint ϕ holds:

$$CSS(\phi) = \{x \in W_X \mid \phi(x)\}.$$

Given a valuation w_X of the random variables in X, we indicate with

$$\mathit{satisfiable}(w_X)$$

the set of constraints that are satisfiable. Each atom in the Herbrand base B_P of R is a Boolean random variable, and there is a countable number of them. The sample space W_R is defined as:

$$W_R = \prod_{a \in B_P} \{true, false\}.$$

According to [112], the sample space W_R is countable, and so the event space of the logic part of the program is defined as $\Omega_R = \mathcal{P}(W_R)$ (powerset of the sample space). However, using Cantor's diagonal argument it is possible to prove that W_R is uncountable, since there is no one-to-one correspondence between the elements of W_R and the set $\mathbb{N}$ of natural numbers, as shown by the following theorem from [135, 136].

Theorem 4. *W_R is uncountable.*

Proof. If the program contains at least one function symbol and one constant, the Herbrand base B_P is denumerable. In this case, each element of W_R can be represented as a denumerable sequence of Boolean values. Equivalently, we can represent it with a denumerable sequence of bits $b_1, b_2, b_3, \ldots$

If we suppose that W_R is denumerable, it is possible to write its element in a list such as

$$\begin{array}{l} b_{1,1}, b_{1,2}, b_{1,3}, \ldots \\ b_{2,1}, b_{2,2}, b_{2,3}, \ldots \\ b_{3,1}, b_{3,2}, b_{3,3}, \ldots \\ \ldots \end{array}$$

Since W_R is denumerable, the list should contain all its elements.

If we pick element $\neg b_{1,1}, \neg b_{2,2}, \neg b_{3,3}, \ldots$, this belongs to W_R because it is a denumerable sequence of Booleans. However, it is not in the list, because it

differs from the first element in the first bit, from the second element in the second bit, and so on. In other words, it differs from each element of the list. This is against the hypothesis that the list contains all elements of W_R. Thus, W_R is not denumerable and so W_R is uncountable. $\square$

The sample space W_P of the entire theory is the Cartesian product of the sample spaces W_X and W_R, $W_P = W_X \times W_R$, and the event space Ω_P is the tensor product (see Definition 4) of the event spaces Ω_X and Ω_R, $\Omega_P = \Omega_X \otimes \Omega_R$. An element w_X of the sample space W_X uniquely determines which constraints are true. We assume that the logic part of the theory $R \cup \mathit{satisfiable}(w_X)$ has a unique well-founded model, denoted with $\mathit{WFM}(w_X)$. Finally, we can extend the probability measure μ_X to a probability measure of the entire theory μ_P since knowing which constraints are true uniquely identifies the truth values of all atoms in the theory. The probability measure on the event space of the whole theory is defined as

$$\mu_P(\omega) = \mu_X(\{w_X \mid (w_X, w_R) \in \omega,\, \mathit{WFM}(w_X) \models w_R\})$$

and the probability of a query q as

$$P(q) = \mu_P(\{(w_X, w_R) \in W_P \mid w_R \models q\}).$$

The authors of [112] (pp. 11-12) say:

> We further know that the event defined by the equation above is an element of the event space Ω_P, since we do not put any restrictions on values of random variables and the event space concerning the logic atoms is defined as the powerset of the sample space [...] thus each subset of the sample space is in the event space.

However, as we have showed, the event space of the logic atoms cannot be defined as the powerset of the sample space, so the fact that the set $\{(w_X, w_R) \in W_P \mid w_R \models q\}$ is measurable is not trivial and must be proved.

Let us introduce the syntax and some examples of PCLP. Variables start with an uppercase letter and are written in bold. Discrete and continuous

random variables can be introduced with the syntax

$$\mathbf{Variable} \sim \mathit{distribution}$$

For example,

$$\mathbf{Time_comp} \sim \mathit{exp}(2)$$

identifies a continuous random variable **Time_comp** that follows an exponential distribution with parameter 2. Constraints among random variables can be represented in the body of rules by enclosing them in square brackets $\langle\ \rangle$. To see this, consider the following two examples from [112], regarding different scenarios.

Example 6 (Fire on a ship [112]). *A fire breaks out in a compartment of a ship and after 0.75 minutes it propagates to the next compartment. If it is not extinguished after 1.25 minutes, it will breach the hull. With this information, we know for sure that the ship will be saved if the fire is under control within 0.75 minutes. This situation can be represented with the following clause:*

$$\mathit{saved} \leftarrow \langle \mathbf{Time_comp_1} < 0.75 \rangle$$

In detail, the previous line states that the value of the continuous random variable $\mathbf{Time_comp_1}$ *should be less than 0.75 for saved to be true. The second compartment is more fragile than the first one, and the fire must be extinguished within 0.625 minutes. However, to reach the second compartment, the fire in the first one must be under control. This means that both fires must be extinguished in* $0.75 + 0.625 = 1.375$ *minutes. In the second compartment, four people can work simultaneously, since it is not as isolated as the first one, meaning that the fire will be extinguished four times faster. We can encode this scenario with:*

$$\begin{aligned} \mathit{saved} \leftarrow & \langle \mathbf{Time_comp_1} < 1.25 \rangle, \\ & \langle \mathbf{Time_comp_1} + 0.25 \cdot \mathbf{Time_comp_2} < 1.375 \rangle \end{aligned}$$

We suppose that the time required to extinguish the fire for both compartments

is exponentially distributed with parameter 1:

$$\mathbf{Time_comp_1} \sim exp(1)$$
$$\mathbf{Time_comp_2} \sim exp(1)$$

The goal is to compute the probability that the ship is saved, i.e., P(saved).

Example 7 (Fruit selling [112])**.** *We want to compute the probability that a consumer buys a certain fruit. We consider two fruits, apples and bananas, whose prices depend on their yields and are modelled with Gaussian distributions.*

$$\mathbf{Yield}(apple) \sim gaussian(12000.0, 1000.0)$$
$$\mathbf{Yield}(banana) \sim gaussian(10000.0, 1500.0)$$

Discrete random variables indicate whether the government supports or not the market:

$$\mathbf{Support}(apple) \sim \{0.3 : yes,\ 0.7 : no\}$$
$$\mathbf{Support}(banana) \sim \{0.5 : yes,\ 0.5 : no\}$$

Two linear functions model the basic price of the fruits, which depends on the yields:

$$basic_price(apple) \leftarrow$$
$$\langle \mathbf{Basic_price}(apple) = 250 - 0.007 \times \mathbf{Yield}(apple) \rangle$$
$$basic_price(banana) \leftarrow$$
$$\langle \mathbf{Basic_price}(banana) = 200 - 0.006 \times \mathbf{Yield}(banana) \rangle$$

Constraints of the form $\langle \mathbf{Variable} = Expression \rangle$ *give a name to an expression involving random variables that can be reused afterwards in other constraints. In fact, the density for* **Variable** *is completely determined by that of the variables in Expression, so there is no need to specify it. The actual price is computed from the basic price plus a fixed amount in case of government*

support:

$$price(Fruit) \leftarrow basic_price(Fruit),$$
$$\langle \mathbf{Price}(Fruit) = \mathbf{Basic_price}(Fruit) + 50 \rangle, \langle \mathbf{Support}(Fruit) = yes \rangle$$
$$price(Fruit) \leftarrow basic_price(Fruit),$$
$$\langle \mathbf{Price}(Fruit) = \mathbf{Basic_price}(Fruit) \rangle, \langle \mathbf{Support}(Fruit) = no \rangle$$

The variable $Fruit$ is not bold, since it is a logical variable, and not a random variable.
A customer buys a certain fruit if its price is below a threshold:

$$buy(Fruit) \leftarrow price(Fruit), \langle \mathbf{Price}(Fruit) \leq \mathbf{Max_price}(Fruit) \rangle$$

The maximum price follows a gamma distribution:

$$\mathbf{Max_price}(apple) \sim \Gamma(10.0, 18.0)$$
$$\mathbf{Max_price}(banana) \sim \Gamma(12.0, 10.0)$$

We can now ask for the probability that a customer buys apples or bananas, represented with $P(buy(apple))$ or $P(buy(banana))$.

These two examples illustrate the expressivity of PCLP, but they do not contain function symbols. The set of random variables is finite, and a semantics for these types of programs was given in [73]. Function symbols can be used to represent integers: for example, with the functor $s/1$ we can identify the successor of a given number. $s(0)$ identifies the successor of 0 (1), $s(s(0))$ the successor of 1 (2), and so on. In the next two examples, we introduce two probabilistic constraint logic programs that adopt this notation.

Example 8 (Gambling)**.** *Consider a gambling game that repeatedly consists in spinning a roulette wheel and drawing a card from a deck. The card is reinserted in the deck after each play. The player keeps track of the final position of the axis of the wheel (the angle it creates with the geographic east). The game continues until the player does not draw a red card. There are four available prizes that can be won, depending on the position of the wheel and the color of the card: prize a if the card is black and the angle is less than π, prize b if*

the card is black and the angle is greater than π, prize c if the card is red and the angle is less than π and prize d otherwise. We describe the angle of the wheel with a uniform distribution in $[0, 2\pi)$, and the color of the card with a Bernoulli distribution with $P(red) = P(black) = 0.5$. In this program, there is a random variable for every ground instantiation of the (anonymous) variable in the probabilistic facts $Card/1$ and $Angle/1$.

$$
\begin{aligned}
&\mathbf{Card}(_) \sim \{red : 0.5,\ black : 0.5\}\\
&\mathbf{Angle}(_) \sim uniform(0, 2\pi)\\
&prize(0, a) \leftarrow \langle \mathbf{Card}(0) = black\rangle, \langle \mathbf{Angle}(0) < \pi\rangle\\
&prize(0, b) \leftarrow \langle \mathbf{Card}(0) = black\rangle, \langle \mathbf{Angle}(0) \geq \pi\rangle\\
&prize(0, c) \leftarrow \langle \mathbf{Card}(0) = red\rangle, \langle \mathbf{Angle}(0) < \pi\rangle\\
&prize(0, d) \leftarrow \langle \mathbf{Card}(0) = red\rangle, \langle \mathbf{Angle}(0) \geq \pi\rangle\\
&prize(s(X), a) \leftarrow prize(X, _), \langle \mathbf{Card}(X) = black\rangle,\\
&\quad \langle \mathbf{Card}(s(X)) = black\rangle, \langle \mathbf{Angle}(s(X)) < \pi\rangle\\
&prize(s(X), b) \leftarrow prize(X, _), \langle \mathbf{Card}(X) = black\rangle,\\
&\quad \langle \mathbf{Card}(s(X)) = black\rangle, \langle \mathbf{Angle}(s(X)) \geq \pi\rangle\\
&prize(s(X), c) \leftarrow prize(X, _), \langle \mathbf{Card}(X) = black\rangle,\\
&\quad \langle \mathbf{Card}(s(X)) = red\rangle, \langle \mathbf{Angle}(s(X)) < \pi\rangle\\
&prize(s(X), d) \leftarrow prize(X, _), \langle \mathbf{Card}(X) = black\rangle,\\
&\quad \langle \mathbf{Card}(s(X)) = red\rangle, \langle \mathbf{Angle}(s(X)) \geq \pi\rangle\\
&at_least_once_prize_a \leftarrow prize(X, a)\\
&never_prize_a \leftarrow \sim at_least_once_prize_a
\end{aligned}
$$

Given this program, with $P(at_least_once_prize_a)$ we represent the probability that the player wins at least one prize a. Similarly, we can describe the probability that the player never wins the prize a with $P(never_prize_a)$.

Example 9 (Hybrid Hidden Markov Model)**.** *A Hybrid Hidden Markov Model (Hybrid HMM) is the combination of a Hidden Markov Model (HMM, with discrete states) and a Kalman Filter (with continuous states). For every integer time point t the system is in a state $[\mathbf{S}(t), \mathbf{Type}(t)]$ described by a discrete random variable $\mathbf{Type}(t)$ taking values in $\{a, b\}$, and a continuous variable*

*$\mathbf{S}(t)$ taking values in $\mathbb{R}$. At time t, the model emits one value $\mathbf{V}(t) = \mathbf{S}(t) + \mathbf{Obs_err}(t)$, where $\mathbf{Obs_err}(t)$ is an error that follows a probability distribution depending on $\mathbf{Type}(t)$, a or b, and not on time. At time $t' = t+1$, the system moves to a new state $[\mathbf{S}(t'), \mathbf{Type}(t')]$, where $\mathbf{S}(t') = \mathbf{S}(t) + \mathbf{Trans_err}(t)$. Again, $\mathbf{Trans_err}(t)$ is an error that follows a probability distribution that depends on $\mathbf{Type}(t)$. Likewise, $\mathbf{Type}(t')$ depends on $\mathbf{Type}(t)$. We use the random variables **Init** and **TypeInit** to describe the state at time 0. All the random variables, except **Init** and **TypeInit**, are indexed by an integer value (time step). We want to compute the probability that the value emitted at time step 1 is larger than 2 (predicate $ok/0$). As before, there is a random variable for every ground instantiation of the variables in the probabilistic facts.*

$$
\begin{array}{l}
ok \leftarrow kf(2), \langle \mathbf{V}(1) > 2 \rangle \\
kf(N) \leftarrow \langle \mathbf{S}(0) = \mathbf{Init} \rangle, \langle \mathbf{Type}(0) = \mathbf{TypeInit} \rangle, kf_part(0, N) \\
kf_part(I, N) \leftarrow I < N, NextI \text{ } is \text{ } I + 1, \\
\quad trans(I, NextI), emit(I), \\
\quad kf_part(NextI, N) \\
kf_part(N, N) \leftarrow N \neq 0 \\
trans(I, NextI) \leftarrow \\
\quad \langle \mathbf{Type}(I) = a \rangle, \langle \mathbf{S}(NextI) = \mathbf{S}(I) + \mathbf{Trans_err_a}(I) \rangle, \\
\quad \langle \mathbf{Type}(NextI) = \mathbf{Type_a}(NextI) \rangle \\
trans(I, NextI) \leftarrow \\
\quad \langle \mathbf{Type}(I) = b \rangle, \langle \mathbf{S}(NextI) = \mathbf{S}(I) + \mathbf{Trans_err_b}(I) \rangle \\
\quad \langle \mathbf{Type}(NextI) = \mathbf{Type_b}(NextI) \rangle \\
emit(I) \leftarrow \\
\quad \langle \mathbf{Type}(I) = a \rangle, \langle \mathbf{V}(I) = \mathbf{S}(I) + \mathbf{Obs_err_a}(I) \rangle \\
emit(I) \leftarrow \\
\quad \langle \mathbf{Type}(I) = b \rangle, \langle \mathbf{V}(I) = \mathbf{S}(I) + \mathbf{Obs_err_b}(I) \rangle \\
\mathbf{Init} \sim gaussian(0, 1) \\
\mathbf{Trans_err_a}(_) \sim gaussian(0, 2) \\
\mathbf{Trans_err_b}(_) \sim gaussian(0, 4)
\end{array}
$$

$\textbf{Obs_err_a}(_) \sim gaussian(0, 1)$

$\textbf{Obs_err_b}(_) \sim gaussian(0, 3)$

$\textbf{TypeInit} \sim \{a : 0.4, b : 0.6\}$

$\textbf{Type_a}(I) \sim \{a : 0.3, b : 0.7\}$

$\textbf{Type_b}(I) \sim \{a : 0.7, b : 0.3\}$

In the next section, we define a precise semantics for hybrid probabilistic logic programs with function symbols.

8.2 Semantics for Hybrid Programs with Function Symbols

We now introduce a new semantics for PCLP[2] and we prove its well-definedness. These results were presented in [21].

We keep discrete and continuous random variables separated. The former are encoded using probabilistic facts as in ProbLog. Recall that with Boolean probabilistic facts it is possible to encode any discrete random variable (see Section 6.1). We consider that a probabilistic constraint logic program is composed of a set of rules R, a set of Boolean probabilistic facts F, and a countable set of continuous random variables $X = X_1, X_2, \ldots$ Each X_i has an associated $Range_i$ that can be $\mathbb{R}$ or $\mathbb{R}^n$. The rules in R define the truth value of the atoms in the Herbrand base of the program given the values of the random variables. We define the sample space of the continuous random variables as $W_X = Range_1 \times Range_2 \times \ldots$ As previously discussed, the probability spaces of individual variables generate an infinite dimensional probability space (W_X, Ω_X, μ_X). The definition of *probabilistic constraint logic theory follows.*

Definition 23 (Probabilistic constraint logic theory)**.** *A probabilistic constraint logic theory P is a tuple* $(\mathrm{X}, W_X, \Omega_X, \mu_X, Constr, R, F)$ *where:*

- X *is a countable set of continuous random variables* $\{X_1, X_2, \ldots\}$*, where each random variable* X_i *has a non-empty range* $Range_i$*.*

[2]In this thesis, we will often use the terms Hybrid Probabilistic Logic Programming and Probabilistic Constraint Logic Programming interchangeably.

- $W_X = Range_1 \times Range_2 \times \ldots$ *is the sample space.*

- Ω_X *is the event space.*

- μ_X *is a probability measure, i.e.,* (W_X, Ω_X, μ_X) *is a probability space.*

- $Constr$ *is a set of constraints closed under conjunction, disjunction, and negation such that* $\forall \varphi \in Constr$, $CSS(\varphi) \in \Omega_X$*, i.e., such that* $CSS(\varphi)$ *is measurable for all* φ*.*

- R *is a set of rules with logical constraints of the form:*

  $h \leftarrow l_1, \ldots, l_n, \langle \varphi_1(X) \rangle, \ldots, \langle \varphi_m(X) \rangle$ *where* l_i *is a literal for* $i = 1, \ldots, n$, $\varphi_j \in Constr$ *and* $\langle \varphi_j(X) \rangle$ *is called constraint atom for* $j = 1, \ldots, m$*.*

- F *is a set of probabilistic facts.*

This definition differs from the one provided in [112] since we separate discrete and continuous probabilistic facts: X is the set containing continuous variables only, while F is a set of discrete probabilistic facts. The probabilistic facts form a countable set of Boolean random variables $Y = \{Y_1, Y_2, \ldots\}$ with sample space $W_Y = \{(y_1, y_2, \ldots) \mid y_i \in \{0, 1\}, i \in 1, 2, \ldots\}$. The event space is the σ-algebra of set of worlds identified by countable set of countable composite choices: a composite choice $\kappa = \{(f_1, \theta_1, y_1), (f_2, \theta_2, y_2), \ldots\}$ can be interpreted as the assignments $Y_1 = y_1, Y_2 = y_2, \ldots$ if Y_1 is associated to $f_1\theta_1$, Y_2 to $f_2\theta_2$ and so on. Finally, the probability space for the entire program (W_P, Ω_P, μ_P) is the product of the probability spaces for the continuous (W_X, Ω_X, μ_X) and discrete (W_Y, Ω_Y, μ_Y) random variables, which exists in light of the following theorem.

Theorem 5 (Theorem 6.3.1 from [47])**.** *Given two probability spaces* (W_X, Ω_X, μ_X) *and* (W_Y, Ω_Y, μ_Y)*, there exists a unique probability space* (W, Ω, μ)*, called the product space, such that* $W = W_X \times W_Y$, $\Omega = \Omega_X \otimes \Omega_Y$*, and*

$$\mu(\omega_X \times \omega_Y) = \mu_X(\omega_X) \cdot \mu_Y(\omega_Y)$$

for $\omega_X \in \Omega_X$ *and* $\omega_Y \in \Omega_Y$*. Measure* μ *is called the* product measure *of* μ_X *and* μ_Y*, and it is also denoted by* $\mu_X \times \mu_Y$*. Moreover, for any* $\omega \in \Omega$*, we define*

its sections *as*

$$\omega^{(X)}(w_X) = \{w_Y \mid (w_X, w_Y) \in \omega\} \quad \omega^{(Y)}(w_Y) = \{w_X \mid (w_X, w_Y) \in \omega\}.$$

Both $\omega^{(X)}(w_X)$ *and* $\omega^{(Y)}(w_Y)$ *are measurable according to* (W_Y, Ω_Y, μ_Y) *and* (W_X, Ω_X, μ_X) *respectively, i.e.,* $\omega^{(X)}(w_X) \in \Omega_Y$ *and* $\omega^{(Y)}(w_Y) \in \Omega_X$*. Consequently,* $\mu_Y(\omega^{(X)}(w_X))$ *and* $\mu_X(\omega^{(Y)}(w_Y))$ *are real values.*

Measure $\mu = \mu_X \times \mu_Y$ *for every* $\omega \in \Omega$ *also satisfies*

$$\mu(\omega) = \int_{W_Y} \mu_X(\omega^{(Y)}(w_Y))d\mu_Y = \int_{W_X} \mu_Y(\omega^{(X)}(w_X))d\mu_X.$$

Thus, $W_P = W_X \times W_Y$, and $\Omega_P = \Omega_X \otimes \Omega_Y$. We indicate with *satisfiable*(w_X) the set of constraints that are satisfiable given a valuation w_X of the random variables in X, and we say that a world *satisfies* a constraint if the values of the continuous random variables in the worlds satisfy the constraint. Starting from a sample $w = (w_X, w_Y)$ from W_P, a ground normal logic program P_w is defined by:

- The groundings of the rules whose constraint are in the set *satisfiable*(w_X), with the constraints removed from the body.
- The probabilistic facts associated to random variables Y_i with value 1.

The well-founded model of $w \in W_P$, $WFM(w)$, is defined as the well-founded model of P_w, $WFM(P_w)$, and we require that it is two-valued: if this constraint is satisfied for every sample $w \in W_P$, we call the program *sound*.

An *explanation* for a query q of a PCLP is a set of worlds ω_i where the query is true in every element of the set, i.e., $\forall w \in \omega_i,\ w \models q$. If every world in which the query is true belongs to the set, the set is termed *covering*. A pairwise incompatible set $\omega = \bigcup_i \omega_i$ is such that $\forall j \neq k,\ \omega_j \cap \omega_k = \emptyset$. We define the probability of a query q as the measure of a covering set of explanations: $P(q) = \mu_P(\{w \mid w \models q\})$. We are now ready to compute the probability of two of the previously discussed examples.

Example 10 (Pairwise incompatible covering set of explanations for Example 8)**.** *Consider Example 8. We represent the extraction of a black card with* $f_1 =$ *black*(_) : 0.5. $(f_1, \theta, 1)$ *means that the card is black while* $(f_1, \theta, 0)$

means that the card is not black (red). We associate a random variable Y_i *to* $black(s^i(0))$*:* $y_i = 1$ *means that in round* i *a black card was picked. The query at_least_once_prize_a has the mutually disjoint covering set of explanations*

$$\omega^+ = \omega_0^+ \cup \omega_1^+ \cup \dots$$

$$\begin{aligned}
\omega_0^+ = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [0, \pi], y_0 = 1\}\\
\omega_1^+ = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [\pi, 2\pi], y_0 = 1, x_1 \in [0, \pi], y_1 = 1\}\\
\dots
\end{aligned}$$

Similarly, the query never_prize_a has the pairwise incompatible covering set of explanations

$$\omega^- = \omega_0^- \cup \omega_1^- \cup \omega_2^- \cup \omega_3^- \cup \dots$$

with

$$\begin{aligned}
\omega_0^- = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [0, \pi], y_0 = 0\}\\
\omega_1^- = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [\pi, 2\pi], y_0 = 0\}\\
\omega_2^- = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [\pi, 2\pi], y_0 = 1, x_1 \in [0, \pi], y_1 = 0\}\\
\omega_3^- = \{&\{(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0, y_1, \dots),\\
&x_0 \in [\pi, 2\pi], y_0 = 1, x_1 \in [\pi, 2\pi], y_1 = 0\}\\
\dots
\end{aligned}$$

Once we compute the pairwise incompatible covering set of explanations, we can calculate the probability of the query.

Example 11 (Probability of queries for Example 8). *Consider sets* ω_0^+ *and* ω_0^- *from Example 10. From Theorem 5,*

$$\mu(\omega_0^+) = \int_{W_X} \mu_Y(\omega^{(\mathrm{X})}(w_X))d\mu_X = \int_{W_X} \mu_Y(\{w_Y \mid (w_X, w_Y) \in \omega\})d\mu_X$$

and so

$$\begin{aligned}\mu(\omega_0^+) &= \int_0^\pi \mu_Y(\{(y_0, y_1, \ldots) \mid y_0 = 1\})d\mu_X \\ &= \int_0^\pi \frac{1}{2} \cdot \frac{1}{2\pi} dx_1 = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}\end{aligned}$$

since, for the discrete variables, $\mu_Y(\{(y_0, y_1, \ldots) \mid y_0 = 0\}) = \mu_Y(\{(y_0, y_1, \ldots) \mid y_0 = 1\}) = 1/2$*, and* μ_X *is the Lebesgue measure of the set* $[0, \pi]$*. Similarly,*

$$\begin{aligned}\mu(\omega_0^-) &= \int_0^\pi \mu_Y(\{(y_0, y_1, \ldots) \mid y_0 = 0\})d\mu_X \\ &= \int_0^\pi \frac{1}{2} \cdot \frac{1}{2\pi} dx_0 = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}.\end{aligned}$$

The sets ω_i^+ *from Example 10 are pairwise incompatible, so the measure of* ω^+ *can be computed by summing the measures of the* ω_i^+*s. By iteratively applying the previous computations, we can calculate the probability of the query* at_least_once_prize_a *as*

$$\begin{aligned}P(at_least_once_prize_a) &= \frac{1}{4} + \frac{1}{4} \cdot \frac{1}{4} + \frac{1}{4} \cdot \left(\frac{1}{4} \cdot \frac{1}{4}\right) + \ldots \\ &= \frac{1}{4} + \frac{1}{4} \cdot \left(\frac{1}{4}\right) + \frac{1}{4} \cdot \left(\frac{1}{4}\right)^2 + \ldots \\ &= \frac{1}{4} \cdot \frac{1}{1 - \frac{1}{4}} = \frac{1}{4} \cdot \frac{4}{3} = \frac{1}{3}\end{aligned}$$

since the sum represents a geometric series. Similarly, for query never_prize_a*,*

the sets in ω^- *are pairwise incompatible, so its probability can be computed as*

$$\begin{aligned}P(never_prize_a) = & \left(\frac{1}{4}+\frac{1}{4}\right)+\left(\frac{1}{4}+\frac{1}{4}\right)\cdot\frac{1}{4}+ \\ & \left(\frac{1}{4}+\frac{1}{4}\right)\cdot\left(\frac{1}{4}\cdot\frac{1}{4}\right)+\ldots \\ = & \frac{1}{2}+\frac{1}{2}\cdot\left(\frac{1}{4}\right)+\frac{1}{2}\cdot\left(\frac{1}{4}\right)^2+\ldots \\ = & \frac{1}{2}\cdot\frac{1}{1-\frac{1}{4}}=\frac{1}{2}\cdot\frac{4}{3}=\frac{2}{3}\end{aligned}$$

$P(never_prize_a) = 1 - P(at_least_once_prize_a)$, *as expected.*

Example 12 (Pairwise incompatible covering set of explanations for Example 9)**.** *Consider Example 9. We represent the discrete state variable with* $f_1 = type(_) : P$, *where* $(f_1, \theta, 0)$ *means that the filter is of type* a *and* $(f_1, \theta, 1)$ *means that the filter is of type* b. *A covering set of explanations for the query* ok *is:*

$$\omega = \omega_0 \cup \omega_1 \cup \omega_2 \cup \omega_3$$

with

$$\begin{aligned}\omega_0 = \{ & (w_X, w_Y) \mid \\ & w_X = (Init, Trans_err_a(0), Trans_err_a(1), Obs_err_a(1), \ldots), \\ & w_Y = (TypeInit, Type(1), \ldots), \\ & Init + Trans_err_a(0) + Trans_err_a(1) + Obs_err_a(1) > 2, \\ & TypeInit = 0, Type(1) = 0\} \\ \omega_1 = \{ & (w_X, w_Y) \mid \\ & w_X = (Init, Trans_err_a(0), Trans_err_b(1), Obs_err_b(1), \ldots), \\ & w_Y = (TypeInit, Type(1), \ldots), \\ & Init + Trans_err_a(0) + Trans_err_b(1) + Obs_err_b(1) > 2, \\ & TypeInit = 0, Type(1) = 1\} \\ \omega_2 = \{ & (w_X, w_Y) \mid \\ & w_X = (Init, Trans_err_b(0), Trans_err_a(1), Obs_err_a(1), \ldots), \\ & w_Y = (TypeInit, Type(1), \ldots),\end{aligned}$$

$$
\begin{aligned}
&\quad Init + Trans_err_b(0) + Trans_err_a(1) + Obs_err_a(1) > 2,\\
&\quad TypeInit = 1, Type(1) = 0\}\\
\omega_3 = &\{(w_X, w_X) \mid\\
&\quad w_X = (Init, Trans_err_b(0), Trans_err_b(1), Obs_err_b(1), \ldots),\\
&\quad w_Y = (TypeInit, Type(1), \ldots),\\
&\quad Init + Trans_err_b(0) + Trans_err_b(1) + Obs_err_b(1) > 2,\\
&\quad TypeInit = 1, Type(1) = 1\}
\end{aligned}
$$

Example 13 (Probability of queries for Example 9)**.** *Consider the set ω_0 from Example 12. We denote discrete random variables $Type(i)$ with y_i. So, $TypeInit = y_0$ and $Type(1) = y_1$. From Theorem 5,*

$$\mu(\omega_0) = \int_{W_X} \mu_Y(\omega^{(\mathrm{X})}(w_X))d\mu_X = \int_{W_X} \mu_Y(\{w_Y \mid (w_X, w_Y) \in \omega\})d\mu_X.$$

Continuous random variables are independent and normally distributed. If $X \sim gaussian(\mu_X, \sigma_X^2)$, $Y \sim gaussian(\mu_Y, \sigma_Y^2)$, and $Z = X + Y$, then $Z \sim gaussian(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$. We indicate with $\mathcal{N}(x, \mu, \sigma^2)$ the Gaussian probability density function with mean μ and variance σ^2. The measure for ω_0 can be computed as:

$$
\begin{aligned}
\mu(\omega_0) &= \int_{-\infty}^{2} \mu_Y(\{(y_0, y_1, \ldots) \mid y_0 = 0, y_1 = 0\})\ d\mu_X\\
&= \int_{-\infty}^{2} 0.4 \cdot 0.3 \cdot \mathcal{N}(x, 0, 1 + 2 + 2 + 1)\ dx = 0.12 \cdot 0.207 = 0.0248.
\end{aligned}
$$

The values for ω_1, ω_2, and ω_3 can be similarly computed. The probability of ω results:

$$P(\omega) = \mu(\omega_0) + \mu(\omega_1) + \mu(\omega_2) + \mu(\omega_3) = 0.25.$$

To prove that a probability value can be assigned to every query, we need to show that every ground query to every sound program is assigned a probability. Here, we focus only on ground programs, but we allow them to be denumerable. This may seem a restriction, but it is not since the number of groundings of a clause can at most be denumerable if the program has function symbols. We start by introducing some definitions.

Definition 24 (Parameterized two-valued interpretations)**.** *Given a ground probabilistic constraint logic program P with Herbrand base B_P, a* parameterized positive two-valued interpretation *Tr is a set of pairs (a, ω_a) where $a \in B_P$ and $\omega_a \in \Omega_P$ such that for each $a \in B_P$ there is only one such pair (Tr is really a function). Similarly, a* parameterized negative two-valued interpretation *Fa is a set of pairs $(a, \omega_{\sim a})$ where $a \in B_P$ and $\omega_{\sim a} \in \Omega_P$.*

Parameterized two-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall a \in B_P, (a, \omega_a) \in I,\ (a, \theta_a) \in J :\ \omega_a \subseteq \theta_a$. For a set T of parameterized two-valued interpretations, the least upper bound and greatest lower bound always exist and are respectively

$$\text{lub}(T) = \{(a, \bigcup_{I \in T, (a, \omega_a) \in I} \omega_a) \mid a \in B_P\}$$

and

$$\text{glb}(T) = \{(a, \bigcap_{I \in T, (a, \omega_a) \in I} \omega_a) \mid a \in B_P\}.$$

The top element $\top$ is

$$\{(a, W_X \times W_Y) \mid a \in B_P\}$$

and the bottom element $\bot$ is

$$\{(a, \emptyset) \mid a \in B_P\}.$$

Definition 25 (Parameterized three-valued interpretations)**.** *Given a ground probabilistic constraint logic program P with Herbrand base B_P, a* parameterized three-valued interpretation *$\mathcal{I}$ is a set of triples $(a, \omega_a, \omega_{\sim a})$ where $a \in B_P$, $\omega_a \in \Omega_P$, and $\omega_{\sim a} \in \Omega_P$, and such that, for each $a \in B_P$, there is one such triple. ω_a and $\omega_{\sim a}$ are the worlds where a is respectively true and false. A parameterized three-valued interpretation $\mathcal{I}$ is* consistent *if $\forall (a, \omega_a, \omega_{\sim a}) \in \mathcal{I},\ \omega_a \cap \omega_{\sim a} = \varnothing$.*

Parameterized three-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall a \in B_P, (a, \omega_a, \omega_{\sim a}) \in I,\ (a, \theta_a, \theta_{\sim a}) \in J,\ \omega_a \subseteq \theta_a$, and $\omega_{\sim a} \subseteq \theta_{\sim a}$. The least upper bound and greatest lower bound for a set T of parameterized three-valued interpretations always exist and are

respectively

$$\text{lub}(T) = \{(a, \bigcup_{I\in T,(a,\omega_a,\omega_{\sim a})\in I} \omega_a, \bigcup_{I\in T,(a,\omega_a,\omega_{\sim a})\in I,} \omega_{\sim a}) \mid a \in B_P\}$$

and

$$\text{glb}(T) = \{(a, \bigcap_{I\in T,(a,\omega_a,\omega_{\sim a})\in I} \omega_a, \bigcap_{I\in T,(a,\omega_a,\omega_{\sim a})\in I} \omega_{\sim a}) \mid a \in B_P\}.$$

The top element $\top$ is

$$\{(a, W_X \times W_Y, W_X \times W_Y) \mid a \in B_P\}$$

and the bottom element $\bot$ is

$$\{(a, \emptyset, \emptyset) \mid a \in B_P\}.$$

Definition 26 ($OpTrueP^P_{\mathcal{I}}(Tr)$ and $OpFalseP^P_{\mathcal{I}}(Fa)$)**.** *For a ground probabilistic constraint logic program P with rules R and facts F, a parameterized two-valued positive interpretation Tr with pairs (a, θ_a), a parameterized two-valued negative interpretation Fa with pairs $(a, \theta_{\sim a})$, and a parameterized three-valued interpretation $\mathcal{I}$ with triplets $(a, \omega_a, \omega_{\sim a})$, we define $OpTrueP^P_{\mathcal{I}}(Tr) = \{(a, \gamma_a) \mid a \in B_P\}$ where*

$$\gamma_a = \begin{cases} W_X \times \omega_{\{\{(a,\emptyset,1)\}\}} & \text{if } a \in F \\ \bigcup_{a\leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m,\varphi_1,\ldots,\varphi_l\in R}((\theta_{b_1} \cup \omega_{b_1}) \cap \ldots & \\ \cap(\theta_{b_n} \cup \omega_{b_n}) \cap \omega_{\sim c_1} \cap \ldots \cap \omega_{\sim c_m} & \text{if } a \in B_P \setminus F \\ \cap CSS(\varphi_1) \times W_Y \cap \ldots \cap CSS(\varphi_l) \times W_Y) & \end{cases}$$

and $OpFalseP^P_{\mathcal{I}}(Fa) = \{(a, \gamma_{\sim a}) \mid a \in B_P\}$ where

$$\gamma_{\sim a} = \begin{cases} W_X \times \omega_{\{\{(a,\emptyset,0)\}\}} & \text{if } a \in F \\ \bigcap_{a\leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m,\varphi_1,\ldots,\varphi_l\in R}((\theta_{\sim b_1} \cap \omega_{\sim b_1}) \cup \ldots & \\ \cup(\theta_{\sim b_n} \cap \omega_{\sim b_n}) \cup \omega_{c_1} \cup \ldots \cup \omega_{c_m} & \\ \cup(W_X \setminus CSS(\varphi_1)) \times W_Y \cup \ldots & \text{if } a \in B_P \setminus F \\ \cup(W_X \setminus CSS(\varphi_l)) \times W_Y) & \end{cases}$$

Proposition 1 (Monotonicity of $OpTrueP^P_{\mathcal{I}}$ and $OpFalseP^P_{\mathcal{I}}$)**.** *$OpTrueP^P_{\mathcal{I}}$ and*

$OpFalseP^P_{\mathcal{I}}$ are monotonic.

Proof. Here we only consider $OpTrueP^P_{\mathcal{I}}$, since the proof for $OpFalseP^P_{\mathcal{I}}$ can be similarly constructed. Essentially, we have to prove that if $Tr_1 \leq Tr_2$ then $OpTrueP^P_{\mathcal{I}}(Tr_1) \leq OpTrueP^P_{\mathcal{I}}(Tr_2)$. By definition, $Tr_1 \leq Tr_2$ means that

$$\forall a \in B_P, (a, \omega_a) \in Tr_1, (a, \theta_a) \in Tr_2 : \omega_a \subseteq \theta_a.$$

Let (a, ω'_a) be the elements of $OpTrueP^P_{\mathcal{I}}(Tr_1)$ and (a, θ'_a) the elements of $OpTrueP^P_{\mathcal{I}}(Tr_2)$. To prove the monotonicity, we have to prove that $\omega'_a \subseteq \theta'_a$.

If $a \in F$, then $\omega'_a = \theta'_a = W_X \times \omega_{\{\{(a,\emptyset,1)\}\}}$. If $a \in B_P \setminus F$, then ω'_a and θ'_a have the same structure. Since $\forall b \in B_P,\ \omega_b \subseteq \theta_b$, then $\omega'_a \subseteq \theta'_a$. □

The monotonicity property ensures that both $OpTrueP^P_{\mathcal{I}}$ and $OpFalseP^P_{\mathcal{I}}$ have a least fixpoint and a greatest fixpoint. We now define an iterated fixpoint operator and prove its monotonicity.

Definition 27 (Iterated fixed point for probabilistic constraint logic programs)**.** *For a ground probabilistic constraint logic program P and a parameterized three-valued interpretation $\mathcal{I}$, we define $IFPCP^P(\mathcal{I})$ as*

$$\begin{aligned} IFPCP^P(\mathcal{I}) = \{(a, \omega_a, \omega_{\sim a}) \mid (a, \omega_a) \in \text{lfp}(OpTrueP^P_{\mathcal{I}}), \\ (a, \omega_{\sim a}) \in \text{gfp}(OpFalseP^P_{\mathcal{I}})\}. \end{aligned}$$

Proposition 2 (Monotonicity of $IFPCP^P$)**.** *$IFPCP^P$ is monotonic.*

Proof. As before, we have to prove that, if $\mathcal{I}_1 \leq \mathcal{I}_2$, then $IFPCP^P(\mathcal{I}_1) \leq IFPCP^P(\mathcal{I}_2)$. By definition, $\mathcal{I}_1 \leq \mathcal{I}_2$ means that

$$\forall a \in B_P, (a, \omega_a, \omega_{\sim a}) \in I_1, (a, \theta_a, \theta_{\sim a}) \in I_2 : \omega_a \subseteq \theta_a, \omega_{\sim a} \subseteq \theta_{\sim a}.$$

Let $(a, \omega'_a, \omega'_{\sim a})$ be the elements of $IFPCP^P(\mathcal{I}_1)$ and $(a, \theta'_a, \theta'_{\sim a})$ the elements of $IFPCP^P(\mathcal{I}_2)$. We have to prove that $\omega'_a \subseteq \theta'_a$ and $\omega'_{\sim a} \subseteq \theta'_{\sim a}$. This is a direct consequence of the monotonicity of $OpTrueP^P_{\mathcal{I}}$ and $OpFalseP^P_{\mathcal{I}}$ in $\mathcal{I}$ proved in Proposition 1. □

$IFPCP^P$ is monotonic and so it has a least fixpoint. We identify $\text{lfp}(IFPCP^P)$ with $WFMP(P)$. We call *depth* of P the smallest ordinal δ such that $IFPCP^P \uparrow \delta = WFMP(P)$. Now we prove that $OpTrueP^P_{\mathcal{I}}$ and $OpFalseP^P_{\mathcal{I}}$ are sound.

Lemma 1 (Soundness of $OpTrueP_{\mathcal{I}}^{P}$)**.** *For a ground probabilistic constraint logic program P with probabilistic facts F, rules R, and a parameterized three-valued interpretation $\mathcal{I}$, denote with θ_a^α the set associated to atom a in $OpTrueP_{\mathcal{I}}^{P} \uparrow \alpha$. For every atom a, world w, and iteration α:*

$$w \in \theta_a^\alpha \to WFM(w \mid \mathcal{I}) \models a$$

where $w \mid \mathcal{I}$ is obtained by adding to w the atoms a for which $(a, \omega_a, \omega_{\sim a}) \in \mathcal{I}$ and $w \in \omega_a$, and by removing all the rules with a in the head for which $(a, \omega_a, \omega_{\sim a}) \in \mathcal{I}$ and $w \in \omega_{\sim a}$.

Proof. We prove the lemma by transfinite induction: we assume that the thesis is true for all ordinals $\beta < \alpha$ and we prove it for α. There are two cases to cover: α successor ordinal and α limit ordinal. Consider α a successor ordinal. If $a \in F$ the statement is easily verified. If $a \notin F$, consider $w \in \theta_a^\alpha$ where

$$\begin{aligned}\theta_a^\alpha = & \bigcup_{a \leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m,\varphi_1,\ldots,\varphi_l \in R} ((\theta_{b_1}^{\alpha-1} \cup \omega_{b_1}) \cap \ldots \\ & \cap (\theta_{b_n}^{\alpha-1} \cup \omega_{b_n}) \cap \omega_{\sim c_1} \cap \ldots \cap \omega_{\sim c_m} \\ & \cap CSS(\varphi_1) \times W_X \cap \ldots \cap CSS(\varphi_l) \times W_X).\end{aligned}$$

This means that there is a rule $a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \varphi_1, \ldots, \varphi_l \in R$ such that $w \in \theta_{b_i}^{\alpha-1} \cup \omega_{b_i}$ for $i = 1, \ldots, n$, $w \in \omega_{\sim c_j}$ for $j = 1 \ldots, m$ and $w \models \varphi_k$ for $k = 1, \ldots, l$. By the inductive assumption and because of how $w \mid \mathcal{I}$ is built, $WFM(w \mid \mathcal{I}) \models b_i$, $WFM(w \mid \mathcal{I}) \models \sim c_j$ and $w \models \varphi_k$ so $WFM(w \mid \mathcal{I}) \models a$.

Consider now α a limit ordinal. Then,

$$\theta_a^\alpha = \text{lub}(\{\theta_a^\beta \mid \beta < \alpha\}) = \bigcup_{\beta < \alpha} \theta_a^\beta.$$

If $w \in \theta_a^\alpha$, there must exist a $\beta < \alpha$ such that $w \in \theta_a^\beta$. By the inductive assumption the hypothesis holds. □

Lemma 2 (Soundness of $OpFalseP_{\mathcal{I}}^{P}$)**.** *For a ground probabilistic constraint logic program P with probabilistic facts F and rules R, and a parameterized three-valued interpretation $\mathcal{I}$, denote with $\theta_{\sim a}^\alpha$ the set associated with atom a in the operator $OpFalseP_{\mathcal{I}}^{P} \downarrow \alpha$. For every atom a, world w and iteration α,*

the following holds:

$$w \in \theta^{\alpha}_{\sim a} \rightarrow WFM(w \mid \mathcal{I}) \models \sim a$$

where $w \mid \mathcal{I}$ is built as in Lemma 1.

Proof. As before, we prove the lemma by transfinite induction: we assume that the thesis is true for all ordinals $\beta < \alpha$ and we prove it for α. Again, we need to cover two cases: α successor ordinal and α limit ordinal. Consider α a successor ordinal. If $a \in F$ the statement is easily verified since probabilistic facts do not appear in the head of any rule. If $a \notin F$ consider $w \in \theta^{\alpha}_{\sim a}$ where

$$\begin{aligned}\theta^{\alpha}_{\sim a} = & \bigcap_{a \leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m,\varphi_1,\ldots,\varphi_l \in R} ((\theta^{\alpha-1}_{\sim b_1} \cap \omega_{\sim b_1}) \cup \ldots \\ & \cup (\theta^{\alpha-1}_{\sim b_n} \cap \omega_{\sim b_n}) \cup \omega_{c_1} \cup \ldots \cup \omega_{c_m} \\ & \cup (W_X \setminus CSS(\varphi_1)) \times W_Y \cup \ldots \cup (W_X \setminus CSS(\varphi_l)) \times W_Y).\end{aligned}$$

This means that, for each $a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \varphi_1, \ldots, \varphi_l \in R$ there exists an index i such that $w \in \theta^{\alpha-1}_{\sim b_i} \cap \omega_{\sim b_i}$, or there exists an index j such that $w \in \omega_{c_j}$, or there exists an index k such that $w \not\models \varphi_k$. By the inductive assumption and because of how $w \mid \mathcal{I}$ is built, either $WFM(w \mid \mathcal{I}) \models \sim b_i$, $WFM(w \mid \mathcal{I}) \models c_j$, or $w \not\models \varphi_k$ should hold, so $WFM(w \mid \mathcal{I}) \models \sim a$.

Consider now α a limit ordinal. Then,

$$\theta^{\alpha}_{\sim a} = \text{glb}(\{\theta^{\beta}_{\sim a} \mid \beta < \alpha\}) = \bigcap_{\beta < \alpha} \theta^{\beta}_{\sim a}.$$

If $w \in \theta^{\alpha}_{\sim a}$, for all $\beta < \alpha$ we have that $w \in \theta^{\beta}_{\sim a}$. By the inductive assumption, for all $\beta < \alpha$, if $w \in \theta^{\beta}_{\sim a}$ this implies that $WFM(w \mid \mathcal{I}) \models \sim a$. Since, $w \in \theta^{\beta}_{\sim a}$ for all $\beta < \alpha$, then $WFM(w \mid \mathcal{I}) \models \sim a$. □

The proof of the soundness of $IFPCP^P$ is based on the following two lemmas.

Lemma 3 (Partial evaluation, Lemma 6 from [136])**.** *For a ground normal logic program P and a three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ such that $\mathcal{I} \leq$*

WFM(P), define $P||\mathcal{I}$ as the program obtained from P by adding all atoms $a \in I_T$ and by removing all rules with atoms $a \in I_F$ in the head. Then, $WFM(P) = WFM(P||\mathcal{I})$.

Lemma 4 (Model equivalence)**.** *Given a ground probabilistic constraint logic program P, for every world w and iteration α the following holds:*

$$WFM(w) = WFM(w \mid IFPCP^P \uparrow \alpha).$$

Proof. Let $(a, \omega_a^\alpha, \omega_{\sim a}^\alpha)$ be the elements of $IFPCP^P \uparrow \alpha$. Consider a three-valued interpretation $\mathcal{I}_\alpha = \langle I_T, I_F \rangle$ with $I_T = \{a \mid w \in \omega_a^\alpha\}$ and $I_F = \{a \mid w \in \omega_{\sim a}^\alpha\}$. Then, $\forall a \in I_T$, $WFM(w) \models a$, and $\forall a \in I_F$, $WFM(w) \models\sim a$. Therefore, $\mathcal{I}_\alpha \leq WFM(w)$.

Since $w \mid IFPCP^P \uparrow \alpha = w||\mathcal{I}_\alpha$, by Lemma 3 we get

$$WFM(w) = WFM(w||\mathcal{I}_\alpha) = WFM(w \mid IFPCP^P \uparrow \alpha).$$

□

We now prove the soundness and completeness of the operator $IFPCP^P$.

Lemma 5 (Soundness of $IFPCP^P$)**.** *For a ground probabilistic constraint logic program P with probabilistic facts F and rules R, denote with ω_a^α and $\omega_{\sim a}^\alpha$ the formulas associated with atom a in $IFPCP^P \uparrow \alpha$. For every atom a, world w and iteration α, the following holds:*

$$w \in \omega_a^\alpha \rightarrow WFM(w) \models a. \tag{8.1}$$

$$w \in \omega_{\sim a}^\alpha \rightarrow WFM(w) \models\sim a. \tag{8.2}$$

Proof. The proof is a consequence of Lemma 4: $w \in \omega_a^\alpha$ means that a is a fact in $w \mid IFPCP^P \uparrow \alpha$. Thus, $WFM(w \mid IFPCP^P \uparrow \alpha) \models a$ and $WFM(w) \models a$.

Similarly, $w \in \omega_{\sim a}^\alpha$ means that there are no rules for a in $w \mid IFPCP^P \uparrow \alpha$, so $WFM(w \mid IFPCP^P \uparrow \alpha) \models\sim a$ and $WFM(w) \models\sim a$. □

Lemma 6 (Completeness of $IFPCP^P$)**.** *For a ground probabilistic constraint logic program P with probabilistic facts F and rules R, let ω_a^α and $\omega_{\sim a}^\alpha$ be the sets associated with atom a in $IFPCP^P \uparrow \alpha$. For every atom a, world w and*

iteration α*, we have:*

$$a \in \mathit{IFP}^w \uparrow \alpha \rightarrow w \in \omega_a^\alpha.$$
$$\sim a \in \mathit{IFP}^w \uparrow \alpha \rightarrow w \in \omega_{\sim a}^\alpha.$$

Proof. We prove it by double transfinite induction. If α is a successor ordinal, assume that

$$a \in \mathit{IFP}^w \uparrow (\alpha - 1) \rightarrow w \in \omega_a^{\alpha-1}$$
$$\sim a \in \mathit{IFP}^w \uparrow (\alpha - 1) \rightarrow w \in \omega_{\sim a}^{\alpha-1}$$

Let us perform transfinite induction on the iterations of $\mathit{OpTrue}^w_{\mathit{IFP}^w\uparrow(\alpha-1)}$ and $\mathit{OpFalse}^w_{\mathit{IFP}^w\uparrow(\alpha-1)}$. Consider a successor ordinal δ and assume that

$$a \in \mathit{OpTrue}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \uparrow (\delta - 1) \rightarrow w \in \gamma_a^{\delta-1}.$$
$$a \in \mathit{OpFalse}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \downarrow (\delta - 1) \rightarrow w \in \theta_{\sim a}^{\delta-1}.$$

where $(a, \gamma_a^{\delta-1})$ are the elements of $\mathit{OpTrue}^P_{\mathit{IFPCP}^P\uparrow\alpha-1} \uparrow (\delta - 1)$ and $(a, \theta_{\sim a}^{\delta-1})$ are the elements of $\mathit{OpFalse}^P_{\mathit{IFPCP}^P\uparrow\alpha-1} \downarrow (\delta - 1)$. We now prove that

$$a \in \mathit{OpTrue}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \uparrow \delta \rightarrow w \in \gamma_a^{\delta}.$$
$$a \in \mathit{OpFalse}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \downarrow \delta \rightarrow w \in \theta_{\sim a}^{\delta}.$$

Consider an atom a. If $a \in F$, the previous statement can be easily proved. Otherwise, $a \in \mathit{OpTrue}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \uparrow \delta$ means that there is a rule $a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, c_m, \varphi_1, \ldots, \varphi_l$ such that for all $i = 1, \ldots, n$,

$$b_i \in \mathit{OpTrue}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \uparrow (\delta - 1) \vee b_i \in \mathit{IFP}^w \uparrow (\alpha - 1)$$

for all $j = 1, \ldots, m$, $\sim c_j \in \mathit{IFP}^w \uparrow (\alpha - 1)$, and for all $k = 1, \ldots, l$, $\varphi_k(w) = \mathit{true}$. For the inductive hypothesis, $\forall i : w \in \gamma_{b_i}^{\delta-1} \vee w \in \omega_{b_i}^{\alpha-1}$ and $\forall j : w \in \omega_{\sim c_j}^{\alpha-1}$, so $w \in \gamma_a^\delta$. The proof is similar for $a \in \mathit{OpFalse}^w_{\mathit{IFP}^w\uparrow(\alpha-1)} \downarrow \delta$.

Consider now δ a limit ordinal, so $\gamma_a^\delta = \bigcup_{\mu<\delta} \gamma_a^\mu$, and $\theta_{\sim a}^\delta = \bigcap_{\mu<\delta} \theta_{\sim a}^\mu$.

If $a \in OpTrue^{w}_{IFP^w \uparrow (\alpha-1)} \uparrow \delta$, then there exists a $\mu < \delta$ such that

$$a \in OpTrue^{w}_{IFP^w \uparrow (\alpha-1)} \uparrow \mu.$$

For the inductive hypothesis, $w \in \gamma^{\delta}_{a}$.

If $a \in OpFalse^{w}_{IFP^w \uparrow (\alpha-1)} \downarrow \delta$, then, for all $\mu < \delta$,

$$a \in OpFalse^{w}_{IFP^w \uparrow (\alpha-1)} \downarrow \mu.$$

For the inductive hypothesis, $w \in \theta^{\delta}_{a}$.

Consider now α a limit ordinal. Then, $\omega^{\alpha}_{a} = \bigcup_{\beta<\alpha} \omega^{\beta}_{a}$ and $\omega^{\alpha}_{\sim a} = \bigcup_{\beta<\alpha} \omega^{\beta}_{\sim a}$.

If $a \in IFP^w \uparrow \alpha$, there exists a $\beta < \alpha$ such that $a \in IFP^w \uparrow \beta$. For the inductive hypothesis $w \in \omega^{\beta}_{a}$, so $w \in \omega^{\alpha}_{a}$. The proof is similar for $\sim a$. □

Now we can prove that $IFPCP^P$ is sound and complete.

Theorem 6 (Soundness and completeness of $IFPCP^P$)**.** *For a sound ground probabilistic constraint logic program P, let ω^{α}_{a} and $\omega^{\alpha}_{\sim a}$ be the sets associated with atom a in $IFPCP^P \uparrow \alpha$. For every atom a and world w there is an iteration α_0 such that for all $\alpha > \alpha_0$ we have:*

$$w \in \omega^{\alpha}_{a} \leftrightarrow WFM(w) \models a. \tag{8.3}$$

$$w \in \omega^{\alpha}_{\sim a} \leftrightarrow WFM(w) \models \sim a. \tag{8.4}$$

Proof. The $\rightarrow$ direction of equations 8.3 and 8.4 is proven in Lemma 5. In the other direction, $WFM(w) \models a$ implies that there exists an α_0 such that $\forall \alpha : \alpha \geq \alpha_0,\ IFP^w \uparrow \alpha \models a$. For Lemma 6, $w \in \omega^{\alpha}_{a}$. Similarly, $WFM(w) \models \sim a$ implies that there exists an α_0 such that $\forall \alpha : \alpha \geq \alpha_0 \rightarrow IFP^w \uparrow \alpha \models \sim a$. As before, for Lemma 6, $w \in \omega^{\alpha}_{\sim a}$. □

Finally, we prove that every query for every sound program is well-defined.

Theorem 7 (Well-definedness of the distribution semantics)**.** *For a sound ground probabilistic constraint logic program P, for all ground atoms a, $\mu_P(\{w \mid w \in W_P, w \models a\})$ is well-defined.*

Proof. Let ω^{δ}_{a} and $\omega^{\delta}_{\sim a}$ be the sets associated with atom a in $IFPCP^P \uparrow \delta$, where δ denotes the depth of the program. Since $IFPCP^P$ is sound and complete, $\{w \mid w \in W_P,\ w \models a\} = \omega^{\delta}_{a}$.

Each iteration of $OpTrueP^P_{IFPCP^P\uparrow\beta}$ and $OpFalseP^P_{IFPCP^P\uparrow\beta}$ for all β generates sets belonging to Ω_P, since the set of rules is countable. So $\mu_P(\{w \mid w \in W_P,\ w \models a\})$ is well-defined. □

Moreover, if the program is sound, for all atoms a, $\omega^\delta_a = (\omega^\delta_{\sim a})^c$ holds, where δ is the depth of the program and the superscript c denotes the complement. Otherwise, there would exist a world w such that $w \notin \omega^\delta_a$ and $w \notin \omega^\delta_{\sim a}$. But w has a two-valued well-founded model, so either $WFM(w) \models a$ or $WFM(w) \models\sim a$. In the first case $w \in \omega^\delta_a$ and in the latter $w \in \omega^\delta_{\sim a}$, against the hypothesis.

8.2.1 A Concrete Syntax

`cplint` hybrid programs [15, 136], are yet another possible approach to manage continuous random variables. They allow the definition of probability densities using the syntax:

$$a : Density \leftarrow Body.$$

Here, a is an atom with an argument Var (not explicitly reported in the formula) that follows a probability density *Density*, and *Body* is the body of a normal clause. In practice, if we want to define a Gaussian distribution with mean 0 and variance 1 on a variable called `Var` in atom `a(Var)`, we can write:

```
a(Var):gaussian(Var,0,1).
```

where `gaussian/3` is a predefined density predicate. Other distributions are available, such as uniform, Dirichlet, gamma, etc. We can also impose constraints by using Prolog comparison predicates. Each predicate a/n and function symbol f/n has a signature that specifies which arguments can hold continuous values (and variables), and only these arguments can contain them. Discrete random variables are identified by ground atoms (there is a countable set of them), while continuous random variables are identified both by predicates and ground terms present in atoms with arguments that can hold continuous random variables. Both terms and continuous variables can appear in atoms and clauses. However, the following constraint must hold: in every world of the program, the values of the term variables in a ground atom for a predicate p/n that is true in the world uniquely identify the values of the continuous variables (see Example 14).

We can also introduce new variables based on a formula involving existing continuous random variables using the special predicate `=:=/2`.

The semantics assigns a probability to any ground atom that does not have continuous values as input arguments. Atoms with continuous input arguments have probability 0, since the probability that a continuous random variable takes a specific value is 0. There are some cases where this constraint can be lifted, and they will be analysed later.

`cplint` hybrid programs can be translated into probabilistic constraint logic programs [112] by removing the continuous arguments of a predicate and by expressing constraints with the supported syntax. Inference in `cplint` hybrid programs can be performed using the module MCINTYRE [134] discussed in Section 7.2.

We now introduce some examples.

Example 14 (Game of card)**.** *Consider a game of card where a player repeatedly needs to pick one out of three possible cards (ace of spades, ace of clubs or ace of hearts) and spin a wheel. The game stops when the player picks the ace of hearts or when the axis of the wheel is between 0 and π degrees (approximated to 3.14 for convenience). We use a continuous random variable with uniform density (*`X`*, line 3) to indicate the angle of the wheel. The outcome of this variable is constrained to be above 3.14 in the clauses for the* `pick/2` *predicate:*

```
1  1/3 :: spades(_).
2  1/2 :: clubs(_).
3  angle(_,X) : uniform_dens(X,0,6.28).
4
5  pick(0,spades) :- spades(0),
6     angle(0,V), V > 3.14.
7  pick(0,clubs) :- \+ spades(0), clubs(0),
8     angle(0,V), V > 3.14.
9  pick(0,hearts) :- \+ spades(0), \+ clubs(0),
10    angle(0,V), V > 3.14.
11
12 pick(s(X),spades):- \+ pick(X,hearts),
13    spades(s(X)), angle(s(X),V), V > 3.14.
```

```
14 pick(s(X),clubs):- \+ pick(X,hearts),
15    \+ spades(s(X)), clubs(s(X)), angle(s(X),V),
16    V > 3.14.
17 pick(s(X),hearts):- \+ pick(X,hearts),
18    \+ spades(s(X)), \+ clubs(s(X)),
19    angle(s(X),V), V > 3.14.
20
21 at_least_once_spades :- pick(_,spades).
22 never_spades :- \+ at_least_once_spades.
```

We can compute the probability that the player picks at least one time spades with the query `at_least_once_spades`*. The continuous random variables form a countable set and are represented by the second argument of predicate* `angle(T,X)`*. There is a continuous random variable for each value of* `T` *(0, $s(0)$, $s(s(0))$, ...). The set Y of discrete Boolean random variables is $\{Y_i^c, Y_i^s \mid i = 0, 1, 2, \dots\}$, where Y_i^c (Y_i^s) represents clubs($s^i(0)$) (spades($s^i(0)$)), and y_i^c (y_i^s) are values for Y_i^c (Y_i^s). Similarly, the set X of continuous random variables is $\{X_i \mid i = 0, 1, \dots\}$. Each continuous variable X_i has a range $[0, 2\pi]$, and we denote its value with x_i. We can consider the mutually disjoint covering set of worlds $\omega = \omega_0 \cup \omega_1 \cup \dots$ for the query* `at_least_once_spades` *where:*

$$
\begin{aligned}
\omega_0 = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0^c, y_0^s, y_1^c, y_1^s, \dots), \\
&x_0 \in [\pi, 2\pi], y_0^s = 1\} \\
\omega_1 = \{&(w_X, w_Y) \mid w_X = (x_0, x_1, \dots), w_Y = (y_0^c, y_0^s, y_1^c, y_1^s, \dots), \\
&x_0 \in [\pi, 2\pi], y_0^s = 0, y_0^c = 1, x_1 \in [\pi, 2\pi], y_1^s = 1\} \\
&\dots
\end{aligned}
$$

That is, for ω_0 spades was selected at round 0 ($y_0^s = 1$) and the wheel (x_0) in the same round was in the range $[\pi, 2\pi]$; for ω_1, spades was not selected at round 0 ($y_0^s = 0$), clubs was selected at round 0 ($y_0^c = 0$), the wheel (x_1) was in the range $[\pi, 2\pi]$ at round 0, spades was selected at round $s(0)$ ($y_1^s = 1$), and the wheel (x_1) was in the range $[\pi, 2\pi]$ at round $s(0)$. Similarly happens for the other ω_i. The probability for ω_0 can be computed as [21, 47]:

$$
\begin{aligned}
\mu(\omega_0) &= \int_{\pi}^{2\pi} \mu_Y(\{(y_0^c, y_0^s, y_1^c, y_1^s, \ldots) \mid y_0^c = 1\})\ d\mu_X \\
&= \int_{\pi}^{2\pi} \frac{1}{3} \cdot \frac{1}{2\pi} dx_1 = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}
\end{aligned}
$$

where $\frac{1}{3}$ *is the contribution of the discrete random variable (spades) and* $\frac{1}{2\pi}$ *is the contribution of the continuous one (angle). The probability for the other* ω_i*s can be similarly computed. Overall, considering the limit, we get* $\frac{1}{3} \cdot \frac{1}{2} \cdot \sum_{i=0}^{\infty}(\frac{2}{3} \cdot \frac{1}{2} \cdot \frac{1}{2})^i = \frac{1}{6} \cdot \sum_{i=0}^{\infty}(\frac{1}{6})^i = \frac{1}{6} \cdot \frac{6}{5} = \frac{1}{5}$ *as probability for the query* `at_least_once_spades`.

The next example encodes a Gaussian mixture:

Example 15 (Gaussian mixture 1)**.** *A Gaussian mixture model is composed of a discrete random variable (usually with a categorical distribution) and a set of Gaussian random variables. The mixture encodes the following generative process: first, the value of the discrete variable is sampled. Then, depending on this value, one of the possible Gaussian random variables is selected and a value is sampled for it.*

We can express a Gaussian mixture model with two components as follows[3]*:*

```
1 h : 0.6.
2 heads :- h.
3 tails :- \+ h.
4 g(X) : gaussian(X, 0, 1).
5 h(X) : gaussian(X, 5, 2).
6 mix(X) :- heads, g(X).
7 mix(X) :- tails, h(X).
8 mix :- mix(X), X > 2.
```

`X` *in clauses for* `mix/1` *follows a distribution that is a mixture of two Gaussians, one with mean 0 and variance 1 with probability 0.6, and one with mean 5 and variance 2 with probability 1 - 0.6 = 0.4. If we want to know what is the probability that the value obtained from this mixture is greater than 2, we can ask the query* `mix`*.*

[3] `https://cplint.eu/e/gaussian_mixture.pl`

Here, predicates `g/1`*,* `h/1`*, and* `mix/1` *have a single output argument which can hold continuous variables. Since there are no term variables, each atom for these predicates in a world univocally determines its argument. The predicate* `mix/1` *requires more attention since it is composed of two clauses with different bodies. However, these are mutually exclusive, so in each world only one of them is true. When* `h` *is true,* `heads` *is true and* `X` *follows the distribution specified at line 4; otherwise, when* `h` *is false,* `tails` *is true and the distribution describing* `X` *is the one introduced at line 5. This property is further discussed in Section 8.2.2.*

Example 16 (Gaussian mixture 2, from [81])**.** *There are two machines,* `a` *and* `b`*, that produce two different widgets with a continuous feature. The first machine produces the widget with probability 0.3, the second with probability 0.7. If machine* `a` *produces the widget, the feature is distributed as a Gaussian with mean 2 and variance 1; otherwise, the feature is distributed as a Gaussian with mean 3 and variance 1. The widget is then processed by a third machine that adds a random quantity to the feature distributed as a Gaussian with mean 0.5 and variance 1.5. This scenario can be encoded as*[4]*:*

```
1 machine(a) : 0.3.
2 machine(b) :- \+ machine(a).
3 st(a,Z) : gaussian(Z, 2, 1).
4 st(b,Z) : gaussian(Z, 3, 1).
5 pt(Y) : gaussian(Y, 0.5, 1.5).
6 widget(X) :- machine(M), st(M,Z), pt(Y), X =:=
     Y + Z.
7 ok_widget :- widget(X), X > 1.0.
```

A constraint checks the correctness of the produced widget. With the query `ok_widget` *we can compute the probability that the produced widget is not flawed.*

`X`*,* `Y`*, and* `Z` *are continuous variables, while* `M` *is a term variable. Since* `X` *is a continuous variable, in every world there should be a single value for* `X` *that makes* `widget(X)` *true. Predicate* `widget/1` *is represented by a single clause with two possible groundings for the term variables, one for* `M` *=* `a`*, and one*

[4]https://cplint.eu/e/widget.pl

for `M` = `b`*. In principle, there could be two values for* `X` *in true groundings of* `widget(X)` *but, as in Example 15, the two groundings have mutually exclusive bodies, since in each world either* `machine(a)` *or* `machine(b)` *is true, but not both.*

Consider now this example[5] that can be used to estimate the mean of a Gaussian:

Example 17 (Estimation of the mean of a Gaussian)**.**

```
1 mean(M) : gaussian(M,1,5).
2 value(_,M,X) : gaussian(X,M,2).
3 value(I,X) :- mean(M), value(I,M,X).
```

For every index `I`*, the continuous variable* `X` *is sampled from a Gaussian with variance 2 and mean* `M` *sampled from another Gaussian with mean 1 and variance 5. Given observations for atom* `value(I,X)` *for different values of* `I`*, we can estimate the mean of a Gaussian by querying* `mean(M)`*.*

The first argument of `value/3` *can hold a term variable while its second and third arguments can hold a continuous variable. The second argument is used as a parameter in the probability density of the third argument. The semantics does not allow specifying the parameters of continuous distributions with values computed by the program, as in this case, so the program may not have a well-defined semantics. However, we can consider continuous variables* `M` *and* `X` *as specified by a joint density*[6]*. This example will be further discussed in Section 8.2.2.*

8.2.2 Syntactic Requirements

As already seen in some examples, the semantics introduced in Section 8.2 requires some syntactic constraints to be satisfied to preserve its well-definedness.

If every variable in the head also appears in a positive literal in the body, a (probabilistic) logic program is called range restricted. In this case, answers to queries are always ground instantiation of it [114, 140], and so probabilities

[5] `https://cplint.eu/e/gauss_mean_est.pl`

[6] A Gaussian distribution with a Gaussian distributed mean is still a Gaussian distribution (if the two distributions are independent). However, we discussed this example for illustrative purposes.

are assigned to ground atoms. In some cases, probabilistic facts are not range restricted. For instance, in Example 14, the first line contains the probabilistic fact `1/3 :: spades(_)`. Here, the answers to queries are still ground instantiations provided that, when the fact is called, all the variables are bound to a constant or they appear in a previous positive literal in the body. In Example 14, the variable for the probabilistic fact is bound to $s^i(0)$. This is formalized in the following theorem:

Theorem 8 (Ground queries with well-defined probability)**.** *Given a Probabilistic Logic Program P and a query q, if P is range restricted and all the variables in probabilistic facts appearing in the body of clauses of P are present in a previous positive literal, the answers to q will be ground instantiation of it, with an associated well-defined probability.*

Proof. We prove Theorem 8 by induction. In the base case, the SLDNF resolution consists of only one step: the query q unifies with a deterministic fact, since there is not a previous positive literal in the body that will allow the presence of a probabilistic fact. The program is range restricted, the substitution θ_1 is computed, and the variables in the query are all grounded. Suppose now that the theorem is true at step n. The current set of substitutions is $\{\theta_1, \theta_2, \ldots, \theta_n\}$. There are two possible cases: if the selected literal of the query matches a deterministic fact, substitution θ_{n+1} grounds the literal, as deterministic facts are already ground. If the selected literal of q matches with a probabilistic fact, this fact cannot be the first of the body of the correspondent clause, by assumption: all the variables appearing in the probabilistic fact are already grounded by preceding substitutions, so the literal is ground when called, and a new substitution θ_{n+1} is computed and added to the list. Eventually, all the variables will be grounded and the answer to the query will be a ground instantiation. □

Note that, if variables appear for the first time in a probabilistic fact, the query can eventually be ground, but the probabilistic fact is not ground when it is called, so the semantics is ill-defined. Thus, the order of the terms in clause bodies is fundamental. Consider this example:

```
1 a(1).
2 f(_,X) : uniform_dens(X,0,6).
```

```
3 g0:- a(X), f(X,V), V > 1.
4 g1:- f(X,V), V > 1, a(X).
```

Here, if the query is `g0`, `X` in `f/2` is ground when the probabilistic fact is called, so the semantics is well-defined. However, this does not hold for the query `g1` (even if they have the same terms in the bodies, but in different order), since `X` is not ground when `f/2` is called.

Another key requirement is that the set of random variables must be countable. In other words, every random variable must be associated with a ground logical atom whose discrete input arguments can contain terms and cannot be real values. Consider the following clause from Example 14:

```
1 pick(0,spades) :- spades(0), angle(0,V),
2     V > 3.14.
```

The first argument of `angle/2` (`0`) is ground. This also holds for `angle(s(X),V)` when it is called during the recursion. In this way, the continuous random variable `V` is associated with either 0 or $s^i(0)$: in both cases, it is a ground logical atom, so the semantics is well-defined.

In the Gaussian mixture example (Example 15), the predicate `mix/1` is defined by two clauses with different bodies: as discussed above, the mutual exclusivity of them preserves the well-definedness of the syntax, since the variable `X` is defined in every world by a single distribution. Similarly for Example 16. This property can be automatically verified using clause unfolding: we obtain two clauses with the same head but mutually exclusive bodies, since they have at least one atom in common but in one clause it is positive, in the other it is negated. This consideration can be directly extended to the case of n clauses with the same head defining n possible distributions for the same variable. If the `mix/1` predicate was defined as

```
1 mix(X) :- g(X).
2 mix(X) :- h(X).
```

the program would be ill-defined since the clauses are not mutually exclusive, and there is not a unique distribution for `X`.

Example 17 is still well-defined since the continuous random variable as input is used as a parameter for another distribution, defining a joint probability density. This also holds for multiple continuous variables as input: for

example, we can model the variance of the Gaussian distribution with another random variable, instead of keeping it fixed to 2 (line 2). However, if the value of a continuous random variable is used as variable for another term and not as parameter for another distribution, the semantics is ill-defined. If we modify Example 17 as follows

```
1 value(X) : gaussian(X,1,5).
2 value(_,M) : gaussian(M,2,2).
3 res(M) :- value(X), value(X,M).
```

the value of `X` in the third line came from a Gaussian distribution and it is used as an input variable for `value/2`, but without being a parameter for another distribution. In this case, the semantics is ill-defined, since the continuous variable `M` cannot be uniquely associated with a ground logical term. This means that there can be an uncountable number of continuous random variables. In Example 17, the term variable used as index (`I`) is associated to a ground logical term, while here it is associated with a real value (`X`). If we change the distribution of the variable `X` to a discrete one, this example would be well-defined, since `X` can be associated with a ground logical term.

To see how to compute the probability for queries to program as Example 17, consider this simplification, where we changed the distribution of the variables for ease of calculation:

```
1 angle_a(_,X,Y) : uniform_dens(Y,X,2).
2 angle_b(X) : uniform_dens(X,0,1).
3 success(I) :- angle_b(X), angle_a(I,X,Y),
4    Y < 1.5.
```

Both variables `X` and `Y` follow a uniform distribution: the bounds for the former are fixed, while the lower bound for the latter is sampled from `X`. Together, they define a joint probability distribution, so we consider them as a multivariate random variable indexed by `I` in the sequence of random variables. Their joint density function is given by

$$f_{XY}(x, y) = \frac{1}{1-0} \cdot \frac{1}{2-x} = \frac{1}{2-x}$$

and the probability of the query `success(0)` can be computed as

$$P(success(0)) = P[Y < 1.5] = \int_0^1 \frac{1}{2-x}\left(\int_x^{1.5} 1\ dy\right) dx =$$
$$= \int_0^1 \frac{1.5-x}{2-x}\, dx = 1 - \frac{ln(2)}{2} \approx 0.653.$$

To clarify all the introduced requirements, consider this program:

```
1  ud(_,V) : uniform_dens(V,0,6).
2  1/2 :: a(_).
3  b(X,1):- a(X).
4  b(X,2):- \+a(X).
5  n(1).
6  f0(X):- a(X).
7  f1:- a(1).
8  f2(X):- n(X), a(X).
9  f3:- a(_).
10 f4(X):- b(X,V), a(V),
11 f5:- ud(1,V), V > 2.
```

The answers of the queries `f0(1)`, `f1`, `f2(X)`, and `f5` are ground instantiations since the input term variable for `a/1` and `ud/2` is bounded to `1` for all four. Similarly for `f4(1)`, where `V` can unify with `1` or `2`, depending on the truth value of `a`. Differently, the answers for `f0(_)`, `f3`, and `f4(_)` are not ground instantiations since, for the first two queries, the term variable of the probabilistic fact `a` is not ground and, for the third query the input variable `X` of `b/2` is not ground.

8.3 Conclusions

In this section, we introduced a new semantics for hybrid probabilistic logic programs, together with a concrete syntax and several syntactic requirements needed to preserve the well-definedness. In the next chapter, we go back to programs with only discrete random variables, and we study the problem of abduction in the context of Probabilistic Logic Programming.

Chapter 9

Extending Probabilistic Logic Programming with Abduction

As already discussed in Section 5.3, Abductive Logic Programming (ALP) [87, 88] extends Logic Programming by considering incomplete data. Often data are also noisy and uncertain, and there can be different levels of belief among rules. To manage these scenarios, in the following section we extend Probabilistic Logic Programming with abduction. In this new framework, abducible facts and probabilistic facts coexist, and integrity constraints can be associated with a probability. After introducing some motivating examples, we formally define this new class of programs and the task we want to solve (Section 9.1), together with a practical algorithm (Section 9.1.2). We conclude this chapter with an overview of related work (Section 9.2). The content of this chapter is a novel contribution introduced in [12].

9.1 Probabilistic Abductive Logic Programs

Real world data are often noisy and incomplete. The integration between abduction and probability seems a natural direction to manage these circumstances. Moreover, we can associate probability values to integrity constraints representing how strong the belief is that the constraint is true. Consider these two motivating examples.

Example 18. *Motivating Example 1. Suppose you work in the city center. There are several possible routes to reach your office starting from your apartment. Clearly, you want to avoid traffic and car accidents. You can associate different probabilities of encountering (or not encountering) a car accident in all the possible streets and impose an integrity constraint to state that only one path can be selected (two different routes cannot be travelled simultaneously). The goal is to select the possible set of streets to maximize the probability of not encountering a car accident.*

Example 19. *Motivating Example 2. Suppose you want to study more in depth a natural phenomenon that may happen in a region. Some variables of the model may describe the morphology of the land, while others are related to the possible events that can occur. Some of these events are unlikely to be observed together, due to some geological features. The goal is to find the set of events that better describes (maximizes the probability of) a possible scenario.*

In other words, given a query (conjunction of ground atoms), the goal is to maximize the joint probability distribution of the query and the constraints by selecting the minimal subset of abducible facts to be included in the abductive logic program while preserving the validity of the constraints.

Let us now introduce some formal definitions.

Definition 28 (Probabilistic Integrity Constraint)**.** *A* probabilistic *integrity constraint is an integrity constraint (see Definition 14) with an associated probability, i.e., is a formula of the form*

$$\pi :- \ Body$$

where $Body = b_1, \ldots, b_n$ *and each* b_i *is a logical literal (i.e., a logical atom or the negation of a logical atom), and* $\pi \in]0, 1]$.

Definition 29 (Probabilistic Abductive Logic Program)**.** *A probabilistic abductive logic program is a triple* $(T, \mathcal{IC}, A)$ *where* T *is an LPAD,* $\mathcal{IC}$ *is a (possibly empty) set of (possibly probabilistic) integrity constraints, and* A *is a set of ground atoms, the* abducibles, *that do not appear in the head of a rule of any grounding of* T.

In other words, a general probabilistic abductive logic program is composed of an LPAD, a set of integrity constraints (deterministic, probabilistic, or both), and a set of abducibles. The particular case where the set of integrity constraints is empty will be discussed later. As usual, we consider only sound LPADs. Abducibles are denoted by the functor `abducible`. For example,

```
1 abducible a.
```

states that atom `a` is an abducible atom.

Starting from the triple $(T, \mathcal{IC}, A)$, we can define a distribution over abductive logic programs P as follows: to obtain a world, we first select one head atom for each grounding of each probabilistic clause from T, and then we add or not each grounding of each probabilistic integrity constraint from $\mathcal{IC}$. The probability of the obtained world is computed as the product of the probabilities of the atomic choices for the clauses multiplied by the probability of each grounding of the probabilistic integrity constraints included in the world, and by one minus the probability for each probabilistic integrity constraint not included.

```
1 a:- b,c.
2 a:- d,e.
3
4 b:0.3.
5 abducible c.
6 d:0.6.
7 abducible e.
8 0.1 :- c,e.
```

(a) Program.

w	b	d	:- c,e.	$P(w)$
1	T	T	I	$0.3 \cdot 0.6 \cdot 0.1 = 0.018$
2	T	F	I	$0.3 \cdot 0.4 \cdot 0.1 = 0.012$
3	F	T	I	$0.7 \cdot 0.6 \cdot 0.1 = 0.042$
4	F	F	I	$0.7 \cdot 0.4 \cdot 0.1 = 0.028$
5	T	T	E	$0.3 \cdot 0.6 \cdot 0.9 = 0.162$
6	T	F	E	$0.3 \cdot 0.4 \cdot 0.9 = 0.108$
7	F	T	E	$0.7 \cdot 0.6 \cdot 0.9 = 0.378$
8	F	F	E	$0.7 \cdot 0.4 \cdot 0.9 = 0.252$

(b) Worlds.

Figure 9.1: Example program and its worlds. I and E indicate respectively whether the IC is included (I) or not (E) in each world.

If we consider the program shown in Figure 9.1a, there are two possible alternatives for each of the two probabilistic facts `b` and `d` and for the IC, so there are 8 possible worlds ($2 \times 2 \times 2$) reported in Figure 9.1b.

Given a probabilistic abductive logic program $(T, \mathcal{IC}, A)$ and a set of ground atoms $\Delta \subseteq A$, the joint probability $P(q, \mathcal{IC} \mid \Delta)$ of a query q and the integrity constraints in $\mathcal{IC}$ to be true in $(T, \mathcal{IC}, A)$ given Δ is the sum of the probabilities

of the worlds where Δ is an abductive explanation of q and all constraints are satisfied.

$P(q, \mathcal{IC} \mid \Delta)$ can be computed by marginalization starting from the joint probability of the worlds, the query, and the ICs:

$$P(q, \mathcal{IC} \mid \Delta) = \sum_w P(q, \mathcal{IC}, w \mid \Delta) = \sum_w P(q, \mathcal{IC} \mid w, \Delta) \cdot P(w \mid \Delta).$$

If call P_w the abductive logic program and IC_w the subset of integrity constraints considered in a world w, then

$$P(q, \mathcal{IC} \mid w, \Delta) = \begin{cases} 1 & \text{if } P_w \cup \Delta \models q \text{ and } P_w \cup \Delta \not\models IC_w \\ 0 & \text{otherwise} \end{cases}$$

so

$$P(q, \mathcal{IC} \mid \Delta) = \sum_{w : P_w \cup \Delta \models q \wedge P_w \cup \Delta \not\models IC_w} P(w \mid \Delta).$$

We define the probabilistic abductive logic problem as follows.

Definition 30 (Probabilistic Abductive Problem)**.** *Given a probabilistic abductive logic program* $(T, \mathcal{IC}, A)$ *and a conjunction of ground atoms* q*, the* query*, the probabilistic abductive problem consists in finding a set* $\Delta \subseteq A$*, the* probabilistic abductive explanation*, such that* $P(q, \mathcal{IC} \mid \Delta)$ *is maximized and the explanations in* Δ *are minimal, i.e., solve*

$$\mathrm{least}(\arg\max_{\Delta} P(q, \mathcal{IC} \mid \Delta))$$

where $\arg\max$ *returns the set of all sets of abducibles that maximizes the joint probability of the query and the ICs (there can be more than one set of abducibles if they all induce the same probability), and*

$$\mathrm{least}(I) = \{\Delta \mid \Delta \in I, \nexists \Delta' \in I : \Delta' \subset \Delta\}.$$

That is, the goal is to find the minimal sets of abducibles that maximizes the joint probability of the query and the integrity constraints. We intend minimality in terms of set inclusion. We also say that least *computes the set of not dominated* Δ*, where* Δ *dominates* Δ' *if* $\Delta \subset \Delta'$*. If* $\mathcal{IC} = \emptyset$*, the task reduces*

to $\text{least}(\arg\max_{\Delta} P(q \mid \Delta))$.

9.1.1 Examples

We now introduce several examples to better illustrate the probabilistic abductive problem.

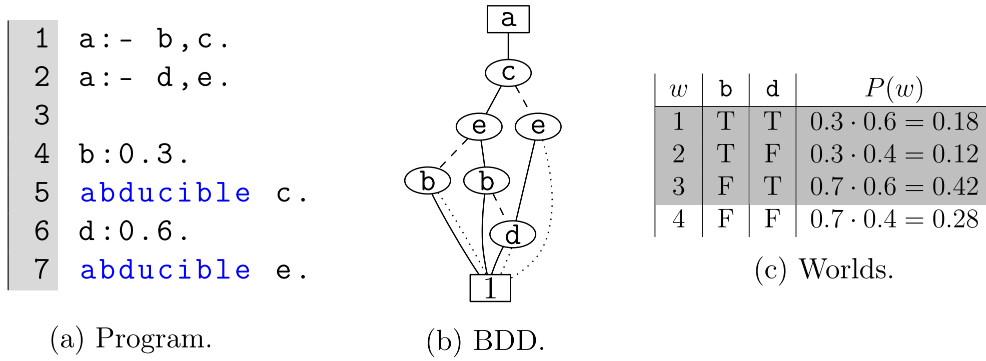
Example 20. *Consider the program shown in Figure 9.1a. The query* $q =$ `a` *has the probabilistic abductive explanation* $\Delta =$ `{c,e}`. $P(q, \mathcal{IC} \mid \Delta) = 0.162 + 0.108 + 0.378 = 0.648$, *corresponding to worlds #5,6,7 of Figure 9.1b, where q is true given Δ and the IC is excluded (E) from the worlds. This happens because the IC does not completely exclude* `{c,e}`, *it just excludes it for the worlds where the constraint is present. This set gives higher probability than* `{e}` *and* `{c}`*:*

- *Given the probabilistic abductive explanation* `{c}`, `a` *is true in 4 worlds (#1,2,5,6) with probability* $0.018 + 0.012 + 0.162 + 0.108 = 0.3$.
- *Given the probabilistic abductive explanation* `{e}`, `a` *is true in 4 worlds (#1,3,5,7) with probability* $0.018 + 0.042 + 0.162 + 0.378 = 0.6$.

Variant 1 *If we remove the integrity constraint from the program shown in Figure 9.1a, as reported in Figure 9.2a, the query* $q =$ `a` *with the probabilistic abductive explanation* $\Delta =$ `{c,e}` *is true in the first three worlds (highlighted in the table of Figure 9.2c) and it has probability* $P(q \mid \Delta) = 0.18 + 0.12 + 0.42 = 0.72$.

Variant 2 *Consider again the program shown in Figure 9.1a, but with a deterministic integrity constraint, i.e.,* `:- c,e`. *There are four possible worlds (see Figure 9.3b). The probabilistic abductive explanation that maximizes the probability of the query* $q =$ `a` *and satisfies the constraint is given by* $\Delta =$ `{e}`.

$P(q, \mathcal{IC} \mid \Delta) = 0.18 + 0.42 = 0.6$, *corresponding to the sum of the probabilities of the worlds where q is true given Δ, highlighted in Figure 9.3b. The probabilistic abductive explanation* `{c,e}` *gives higher probability than* `{e}` *(see above), but is forbidden by the IC.*



```
1  a:- b,c.
2  a:- d,e.
3
4  b:0.3.
5  abducible c.
6  d:0.6.
7  abducible e.
```

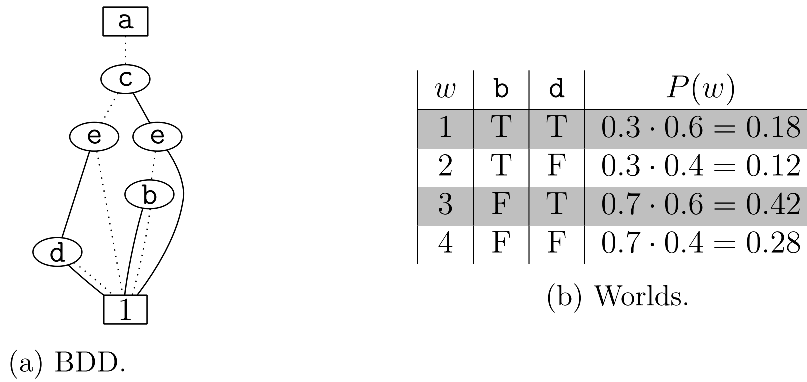
(a) Program.

(b) BDD.

w	b	d	$P(w)$
1	T	T	$0.3 \cdot 0.6 = 0.18$
2	T	F	$0.3 \cdot 0.4 = 0.12$
3	F	T	$0.7 \cdot 0.6 = 0.42$
4	F	F	$0.7 \cdot 0.4 = 0.28$

(c) Worlds.

Figure 9.2: Program, BDD, and worlds for Example 20 variant 1. Highlighted rows in the table represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{c,e}`, together with their probability.



(a) BDD.

w	b	d	$P(w)$
1	T	T	$0.3 \cdot 0.6 = 0.18$
2	T	F	$0.3 \cdot 0.4 = 0.12$
3	F	T	$0.7 \cdot 0.6 = 0.42$
4	F	F	$0.7 \cdot 0.4 = 0.28$

(b) Worlds.

Figure 9.3: BDD and worlds for Example 20 variant 2. Highlighted rows in the table represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{e}`, together with their probability.

Variant 3 *If the probability of the IC is set to 0.5 (*`0.5 :- c,e`*), query* q = `a` *has the probabilistic abductive explanation* Δ = `{e}` *with probability* $P(q, \mathcal{IC} \mid \Delta) = 0.09 \cdot 2 + 0.21 \cdot 2 = 0.6$*, corresponding the worlds #1,3,5,7 (highlighted in Table 9.1). Such explanation gives higher probability than* `{c,e}` *and* `{c}`*:*

- *Given the probabilistic abductive explanation* `{c,e}`*,* `a` *is true in 3 worlds (#5,6,7) with probability* $0.09 + 0.06 + 0.21 = 0.36$.
- *Given the probabilistic abductive explanation* `{c}`*,* `a` *is true in 4 worlds (#1,2,5,6) with probability* $0.09 + 0.06 + 0.09 + 0.06 = 0.3$.

The smallest probability π *of the IC* π`:-c,e.` *such that set* `{e}` *is chosen can*

w	b	d	:- c,e.	$P(w)$
1	T	T	I	$0.3 \cdot 0.6 \cdot 0.5 = 0.09$
2	T	F	I	$0.3 \cdot 0.4 \cdot 0.5 = 0.06$
3	F	T	I	$0.7 \cdot 0.6 \cdot 0.5 = 0.21$
4	F	F	I	$0.7 \cdot 0.4 \cdot 0.5 = 0.14$
5	T	T	E	$0.3 \cdot 0.6 \cdot 0.5 = 0.09$
6	T	F	E	$0.3 \cdot 0.4 \cdot 0.5 = 0.06$
7	F	T	E	$0.7 \cdot 0.6 \cdot 0.5 = 0.21$
8	F	F	E	$0.7 \cdot 0.4 \cdot 0.5 = 0.14$

Table 9.1: Worlds for Example 20 variant 3. Highlighted rows represent the worlds in which the query `a` is true with the probabilistic abductive explanation `{e}`, together with their probability. I and E stand respectively for included and excluded.

be computed by solving a system of two inequalities, imposing that the sum of the probabilities of worlds #5,6,7 (see Figure 9.1b) is greater than the sum of the probabilities associated both with worlds #1,2,5,6 and #1,3,5,7, with π *as a variable. The result is* $\pi < 0.167$*. So, when the IC has probability greater than 0.167, the probabilistic abductive explanation* `{e}` *is preferred to* `{c,e}` *(and* `{c}`*), as if the constraint were deterministic.*

Example 21. *Abducible facts can also be negated in the body of clauses. Consider a variation of the program shown in Figure 9.2, where the abducible* `c` *appears negated in the first clause for* `a/0`*:*

```
a:- b,\+c.
```

Here, the query $q =$ `a` *has the probabilistic abductive explanation* $\Delta =$ `{e}` *with probability 0.72, because, when* `c` *is not selected, the second clause still has the body satisfied.*

Example 22. *A program may have multiple minimal explanations yielding maximum probability of the query and the constraints. Consider the following example:*

```
1 a:0.4.
2 b:0.4.
3 abducible aa.
```

```
4 abducible bb.
5 q:- a,aa.
6 q:- b,bb.
7 :- aa,bb.
```

Both $\Delta_1 =$ `{aa}` *and* $\Delta_2 =$ `{bb}` *are minimal, each one giving a probability of 0.4.*

Example 23. *Consider the case of an abductive logic program (no probabilistic facts). For example, if we query* `a` *in the following program, where both* `b` *and* `c` *are abducibles*

```
1 a:- b,c.
2 a:- c.
3 abducible b.
4 abducible c.
```

we would obtain, without the least *function, two explanations:* $\Delta_1 =$ `{b,c}` *and* $\Delta_2 =$ `{c}`*. However, this is in contrast with our definition, where the goal is to find sets that are also minimal. In this example* $\Delta_2 \subset \Delta_1$*, so the latter must not be considered as it is not minimal.*

Variant 1 *If we add another clause* `a:- d,e.` *with* `d` *and* `e` *abducibles, the set of explanations for* `a` *will be* $\Delta =$ `{{c},{d,e}}`*, since both are minimal.*

We now consider a possible encoding of Example 19 (proposed in Example 5 reported here for clarity).

Example 24. *The island of Stromboli is located at the intersection of two geological faults, one in the southwest-northeast direction, the other in the east-west direction, and contains one of the three volcanoes that are active in Italy. This program, taken from [35, 138], models the possibility that an eruption or an earthquake occurs at Stromboli:*

```
1 eruption:0.6;earthquake:0.3 :- sudden_er,
     fault_rupture(X).
2 sudden_er:0.7.
3 fault_rupture(southwest_northeast).
4 fault_rupture(east_west).
```

If there is a sudden energy release (`sudden_er`*) under the island and there is a fault rupture (*`fault_rupture(X)`*), there can be an eruption of the volcano on the island with probability 0.6 or an earthquake with probability 0.3. A sudden energy release occurs with probability 0.7, and we are sure that ruptures occur along both faults.*

Variant 1 *If we make the two* `fault_rupture/1` *facts abducible*[1]*, the query* $q =$ `eruption` *has the probabilistic abductive explanation*
$\Delta =$ `{fault_rupture(southwest_northeast),fault_rupture(east_west)}` *with probability* $P(q \mid \Delta) = 0.252 + 0.126 + 0.042 + 0.126 + 0.042 = 0.588$*, corresponding to worlds #1,2,3,7,13 in Table 9.3 where q is true given* Δ*.* Δ *gives the highest probability since:*

- *given the probabilistic abductive explanations*
  $\Delta_1 =$ `{fault_rupture(southwest_northeast)}` *or*
  $\Delta_2 =$ `{fault_rupture(east_west)}`*,* $P(q \mid \Delta_1) = P(q \mid \Delta_2) = 0.42$*;*
- *given the probabilistic abductive explanation* $\Delta_3 = \emptyset$*,* $P(q \mid \Delta_3) = 0$*.*

Variant 2 *Given the program:*

```
1 eruption:0.6; earthquake:0.3 :- sudden_er,
     fault_rupture(_).
2 sudden_er: 0.7.
3 abducible fault_rupture(southwest_northeast).
4 fault_rupture(east_west).
```

the query $q =$ `eruption` *has the probabilistic abductive explanation*
$\Delta =$ `{fault_rupture(southwest_northeast)}` *with the same probability as above, corresponding to the same worlds. The same result would be achieved by making abducible* `fault_rupture(east_west)` *instead of* `fault_rupture(southwest_northeast)`*.*

Variant 3 *If we remove line 3 or line 4 from the program, for instance line 4:*

[1]This variant can be tested at `https://cplint.eu/e/eruption_abduction.pl`

w	eruption:0.6; earthquake:0.3 :- sudden_er, fault_rupture(sw_ne).	sudden_er:0.7.	$P(w)$
1	eruption	sudden_er	0.42
2	eruption	*null*	0.18
3	earthquake	sudden_er	0.21
4	earthquake	*null*	0.09
5	*null*	sudden_er	0.07
6	*null*	*null*	0.03

Table 9.2: Possible worlds for the LPAD of Example 24 (Variant 3) with the corresponding probability, computed as the product of the probabilities associated with the head atoms taking value true, reported in each row. Highlighted rows represent the worlds in which the query `eruption` is true.

```
1 eruption:0.6; earthquake:0.3 :- sudden_er,
     fault_rupture(_).
2 sudden_er: 0.7.
3 abducible fault_rupture(southwest_northeast).
```

we would lose the second grounding `X/east_west`*. Now, the query* $q =$ `eruption` *has the probabilistic abductive explanation*
$\Delta =$ `{fault_rupture(southwest_northeast)}` *with probability* $P(q \mid \Delta) = 0.42 + 0.18 = 0.6$*, corresponding to worlds #1,2 in Table 9.2, where* q *is true given* Δ*.*

Variant 4 *If we add an IC to the program stating that a fault rupture cannot happen along both directions at the same time:*

```
1 eruption:0.6; earthquake:0.3 :- sudden_er,
     fault_rupture(_).
2 sudden_er: 0.7.
3 abducible fault_rupture(southwest_northeast).
4 abducible fault_rupture(east_west).
5
6 :- fault_rupture(southwest_northeast),
     fault_rupture(east_west).
```

the probabilistic abductive explanations that maximize the probability of the

w	eruption:0.6; earthquake:0.3 :- sudden_er, fault_rupture(sw_ne).	eruption:0.6; earthquake:0.3 :- sudden_er, fault_rupture(east_west).	sudden_er:0.7.	$P(w)$
1	eruption	eruption	sudden_er	0.252
2	eruption	earthquake	sudden_er	0.126
3	eruption	*null*	sudden_er	0.042
4	eruption	eruption	*null*	0.108
5	eruption	earthquake	*null*	0.054
6	eruption	*null*	*null*	0.018
7	earthquake	eruption	sudden_er	0.126
8	earthquake	earthquake	sudden_er	0.063
9	earthquake	*null*	sudden_er	0.021
10	earthquake	eruption	*null*	0.054
11	earthquake	earthquake	*null*	0.027
12	earthquake	*null*	*null*	0.009
13	*null*	eruption	sudden_er	0.042
14	*null*	earthquake	sudden_er	0.021
15	*null*	*null*	sudden_er	0.007
16	*null*	eruption	*null*	0.018
17	*null*	earthquake	*null*	0.009
18	*null*	*null*	*null*	0.003

Table 9.3: Possible worlds for the LPAD of Example 24 with the corresponding probability computed as the product of the probabilities associated with the head atoms taking value true, reported in each row. Highlighted rows represent the worlds in which the query `eruption` is true.

query q = `eruption` *and satisfy the constraint are both*
Δ_1 = `{fault_rupture(southwest_northeast)}`
and
Δ_2 = `{fault_rupture(east_west)}`, *as* $P(q, \mathcal{IC} \mid \Delta_1) = P(q, \mathcal{IC} \mid \Delta_2) = 0.252 + 0.126 + 0.042 = 0.42$. *The probabilistic abductive explanation found at the beginning of this example, yielding a higher probability* $P(q \mid \Delta) = 0.588$, *is now forbidden by the IC.*

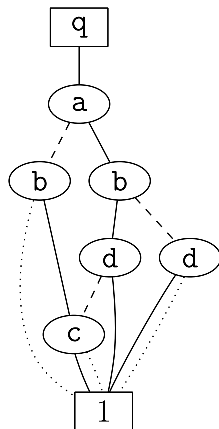
9.1.2 Algorithm

We exploit BDDs to solve the probabilistic abductive problem. Integrity constraints are implemented by conjoining BDDs. We can obtain a BDD for an IC of the form $:- b_1, \ldots, b_n$ by asking with PITA the query $b_1, \ldots, b_n$ (after applying the PITA transformation, see [136]). If the IC has an associated prob-

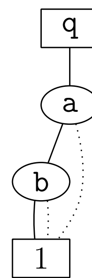
ability, an extra variable is added to the BDD representing it. Some nodes of the BDD are marked to represent abducible facts. Then, two BDDs are built, one for the query (BDDQ) and one for the constraint (BDDC). The Boolean expression representing the query is given by the conjunction of BDDQ with the negation of BDDC (BDDQ and not BDDC). In case of multiple ICs, this operation can be straightforwardly extended. If we consider the following program:

```
1 q:- a,d.
2 q:- b,c.
3 abducible a.
4 abducible b.
5 c:0.4.
6 d:0.5.
7 :- a,b.
```

with the query `q`, BDDQ (Figure 9.4a) represents the Boolean expression `(a and d) or (b and c)` while BDDC (Figure 9.4b) represents `a and b`. The two BDDs are combined to obtain: `((a and d) or (b and c)) and (not(a and b))` (Figure 9.5a) whose truth table is reported in Table 9.5b.



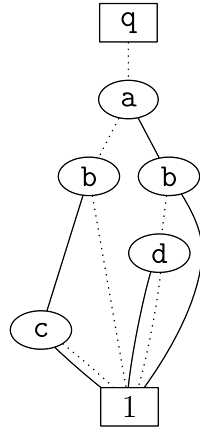
(a) BDD for `(a and d) or (b and c)` (BDDQ).



(b) BDD for `a and b` (BDDC).

Figure 9.4: BDDs for the example showing the conjunction of BDDs.

We develop algorithms 5 and 6 to solve the probabilistic abductive problem. Here, we focus on programs without function symbols. In detail, the function



(a) BDD resulting from the conjunction of BDDQ and BDDC.

a	b	c	d	Expr
F	F	F	F	F
F	F	F	T	F
F	F	T	F	F
F	F	T	T	F
F	T	F	F	F
F	T	F	T	F
F	T	T	F	T
F	T	T	T	T
T	F	F	F	F
T	F	F	T	T
T	F	T	F	F
T	F	T	T	T
T	T	F	F	F
T	T	F	T	F
T	T	T	F	F
T	T	T	T	F

(b) Truth table.

Figure 9.5: BDD and truth table for the example showing the conjunction of BDDs. Highlighted rows represent the combinations of arguments such that the expression `((a and d) or (b and c)) and (not(a and b))` (compactly referred as Expr in the table) is true.

Algorithm 5 Function AbductiveExpl: computation of the *minimal* sets that maximize the joint probability of the query and the ICs, and of the corresponding probability.

1: **function** AbductiveExpl($root$)
2: $root' \leftarrow$ Reorder($root$) ▷ BDD reordering
3: $TableProb \leftarrow \emptyset$
4: $TableAbd \leftarrow \emptyset$
5: $(Prob, Abd) \leftarrow$ AbdInt($root'$, $TableProb$, $TableProb$, $false$)
6: $Abd' \leftarrow$ RemoveDominated(Abd)
7: **return** $(Prob, Abd')$
8: **end function**

AbductiveExpl (Algorithm 5) receives as input the root of the BDD representing the explanations for the query. This BDD is reordered so that variables associated with abducibles come first in the order: this operation may vary the size of the BDD, but it is crucial to have nodes representing abducible first so that Algorithm 1 can be directly applied. Table *TableAbd* stores the set of explanations together with the associated probabilities computed at each node representing an abducible. *TableProb* stores the probability values computed at nodes representing probabilistic facts and is used by the function Prob of Algorithm 1. Both are initially empty.

Algorithm 6 Function AbdInt: computation of the sets that maximize the joint probability of the query and the ICs, and of the corresponding probability, through BDD exploration.

```
1: function AbdInt(node, TableProb, TableProb, comp)
2:     comp ← node.comp ⊕ comp
3:     if var(node) is not associated to an abducible then
4:         p ←Prob(node)                                         ▷ Call to prob
5:         if comp then
6:             return (1 − p, [[]])
7:         else
8:             return (p, [[]])
9:         end if
10:    else
11:        if TableAbd(node.pointer) ≠ null then
12:            return TableAbd(node.pointer)
13:        else
14:            (p_0, Abd_0) ← AbdInt(child_0(node), TableProb, TableAbd, comp)
15:            (p_1, Abd_1) ← AbdInt(child_1(node).TableProb, TableAbd, comp)
16:            if p_1 > p_0 then                                  ▷ Max
17:                Abd ← AddNodeToExplanations(var(node), Abd_1)
18:                Res ← (p_1, Abd)
19:            else if p_1 == p_0 then                            ▷ Same probability
20:                Abd ←RemoveDominatedAndMerge(Abd_0,Abd_1)
21:                if Abd is empty then
22:                    Res ← (p_0, Abd_0)
23:                else
24:                    Res ← (p_1, Abd)
25:                end if
26:            else
27:                Res ← (p_0, Abd_0)
28:            end if
29:            Add node.pointer → Res to TableAbd
30:            return Res
31:        end if
32:    end if
33: end function
```

After the reordering of the BDD, the function AbdInt from Algorithm 6 is called: starting from the root of the BDD, if the current node does not represent an abducible (i.e., it is the terminal node or a probabilistic node), there are no more nodes associated with abducibles in the lower levels, thanks to the

reordering operation, so the function PROB is called and a set containing only the empty explanation is returned. In case of a node representing an abducible, if a value for the current node has already been computed, it is retrieved from *TableAbd*. If it is not, the function ABDINT is recursively called on the true and false child. After the recursion, a *max* operation (line 16) selects the child with the highest associated probability: if the probability of the true child is greater than the probability of the false child, the abducible represented by the current node is selected and added to the current explanations. Otherwise, it is not. If it is selected (line 18), the probability at the current node is given by the probability of the true child (p_1). Moreover, the function ADDNODETO-EXPLANATIONS builds the set of explanations as the union between the set of the true child choices (Abd_1) together with the current abducible ($var(node)$). If the probabilities of the true and false child are the same, we remove (line 20) the explanations of the true child that are dominated (strict superset) by an explanation of the false child. This operation is needed to preserve the minimality of the result, otherwise we would obtain sets of explanations that are not minimal. Removing explanations of the false child that are dominated by an explanation of the true child is not needed since, after the introduction of the current node in the explanations for the true child, the explanations that dominate the ones removed in the false child are no more subsets (the current node is not present in the false explanations). If, after the removal of the dominated explanations, the obtained set is empty, the explanations of the false child (Abd_0) and the associated probability p_0 are returned, because the inclusion of the abducible in the explanations of the true child would still lead to a dominated explanation. Function REMOVEDOMINATEDANDMERGE adds the current node to all the explanations of Abd_1 and merges them with Abd_0. To speed up the comparisons, the sets of explanations are kept ordered.

The variable *Res* will be associated with the pair $(Prob, Abd)$ where $Prob = P(q, \mathcal{IC} \mid \Delta)$ and *Abd* is the set of explanations maximizing that value. Intermediate *Res* results are stored in *TableAbd* to avoid recomputing them when encountering an already visited node.

Once the function ABDINT returns, we remove once again the possible dominated sets from the set of explanations (Algorithm 5 line 6). Finally, Algorithm 5 returns the pair $(Prob, Abd')$ with $Prob = P(q, \mathcal{IC} \mid \Delta)$ and

$Abd' = \text{least}(\arg\max_{\Delta} P(q, \mathcal{IC} \mid \Delta))$ (the set of minimal sets Δ maximizing that probability).

In the case where the set of probabilistic facts is empty, Algorithm 5 returns the abductive explanations: a BDD encodes a Boolean function that is a solution for the abductive problem and, in case of multiple solutions, the dominated ones are removed by the functions REMOVEDOMINATEDANDMERGE and REMOVEDOMINATED. Thus, the returned solutions are minimal.

To study the complexity of the task, we need to consider that exact inference in probabilistic logic programs is #P-complete, as already discussed in Section 7.1. The procedure we described here follows the same steps of exact inference in PLP, consisting of knowledge compilation and traversal of the obtained structure using a dynamic programming algorithm. Here, we have an additional step that consists in reordering the variables of the BDD. This can be performed polynomially (Section 7.1). In the experiments (see Section 9.1.3) we empirically noticed that the execution time spent to reorder the BDD is negligible with respect to its traversal. Checking the subset relation between two sets can be performed in linear time with respect to the size of the smallest one, if these are ordered (as in our case). If the sets of explanations are of sizes respectively m and n, $m \cdot n$ comparisons are required.

To clarify the process, consider Example 20 variant 1 and its BDD representation (Figure 9.2c). The algorithm starts at node `a` and it is recursively called until a node not representing an abducible is found. Nodes `b` left and right are reached, and the function PROB returns 0.3 for `b` left and $0.3 + (1 - 0.3) \cdot 0.6 = 0.72$ for `b` right. At the left `e` node, $\max(0.3,0.72) = 0.72$ and `e` is added to the current (empty) explanation. Similarly, at node `c` $\max(0.72,0.6) = 0.72$, so `c` is added to the true child explanation `{e}`. The computed probability is 0.72 corresponding to the set of explanations `{c,e}`.

With the following theorem, we prove that algorithms 5 and 6 together solve the probabilistic abductive problem.

Theorem 9. *Algorithm 5 solves the probabilistic abductive problem.*

Proof. (Sketch) The BDDs for the query and the ICs represent the Boolean formulas according to which the query is true and the ICs are satisfied for the correctness of the PITA algorithm. By reordering the resulting BDD, we have

abducible nodes first in the diagram: this means that, when we reach a probabilistic node, there are no more abducible nodes below, and we can compute the probability of that node as in PITA. The upper diagram is used to select the sets of abducibles that provide the largest probability by simply comparing the probabilities of the partial sets coming from the children. Special care must be taken for the case of equal probability of the two children because in this case domination must be checked. □

9.1.3 Experiments

To test our algorithm, we conducted several experiments on the same machine described in Section 7.2.1. We selected five synthetic datasets[2] taken from [32]: growing head (gh), growing negated body (gn), blood, probabilistic graph (graph) and probabilistic complete graph (complete graph). For each of the five, we conducted three experiments: one with deterministic integrity constraints, one with probabilistic integrity constraints, and one without integrity constraints. The results for the first two are almost identical, so in the plots only one curve is shown. We set the probability for the probabilistic integrity constraint to 0.5. This value usually indicates weak constraints, but here we are interested in analysing the execution time, not in the computed probability. With a different probability value for the constraints we would get almost the same results in terms of execution time, since the BDD must be traversed in the same manner.

We selected the five previously listed datasets with the goal of covering a broad spectrum of applications: with gh and gnb we investigate how a growing number of atoms in the head and negated literals in the body influences the execution time. In the gh dataset with integrity constraints, there are multiple explanations with the same probability. The dataset blood represents a possible application in the biological domain, while the experiments on graphs represent the scenario described in Example 18. For all five, we computed the total execution time which is composed of the time required for constructing, reordering, and traversing the BDD. As we discuss in Section 9.2, there does not exist comparable systems to the best of our knowledge, so a direct

[2]All datasets can be found at: `https://bitbucket.org/machinelearningunife/palp_experiments`.

comparison with other solutions is not possible.

Details of the Datasets

The dataset gh is composed of a set of programs where clauses have an increasing number of atoms in the head (from 1 to 14). The most complex program has 28 clauses and 14 abducibles. For example, this is a program with two abducibles:

```
1 abducible aba1.
2 abducible aba2.
3 a0 :- a1.
4 a1:0.5:- aba1.
5 a0:0.5; a1:0.5:- a2.
6 a2:0.5:- aba2.
```

The query is `a0`. For experiments with ICs, we considered an XOR constraint: only one abducible should be selected. If we consider the previous program, this can be implemented as:

```
1 r:- aba1,aba2.
2 r:- \+aba1,\+aba2.
3 :- r.
```

In general, if there are `n` abducibles, an XOR constraint can be implemented with $\binom{n}{2} + 2$ clauses. In the previous example, $\binom{2}{2} + 1 = 3$. The second clause (line 2) represents the case where no abducibles are considered.

The gnb dataset is composed of a set of programs with an increasing number (from 1 to 14) of negated atoms in the body of clauses. Every clause has an abducible in the body. The most complex program has 121 clauses and 16 abducibles. The following is a program with three abducibles:

```
1 abducible aba0.
2 abducible aba1.
3 abducible aba2.
4 a0:0.5:- a1,aba0.
5 a0:0.5:- \+a1,a2,aba0.
6 a1:0.5:- a2,aba1.
7 a2:0.5:- aba2.
```

The goal is to compute the probability of a0. In the experiment with ICs, we tested the edge case where all the abducibles should be selected, a situation that can be represented with:

```
1 r:- \+aba0.
2 r:- \+aba1.
3 r:- \+aba2.
4 :- r.
```

The blood dataset is a set of programs that models the inheritance of blood type. Each program has an increasing number of ancestors (up to five levels in the genealogical tree). The most complex program has 67 clauses and 2 abducibles with a variable argument with 20 possible groundings each. For the experiments with the ICs, we implemented a constraint as a single denial with variables imposing that father and mother should not have the same blood type. The goal is a person p having a certain blood type.

The graph dataset is composed of a set of probabilistic graphs following a Barabási-Albert model. We generated them using the Python networkx package[3], with the number of nodes ranging in $[50, 100]$ with a step of 10 and parameter m_0 (representing the number of edges to attach from a new node to existing nodes) set to 2. The generation of the Barabási-Albert model is not deterministic, so we created 100 different graph configurations and averaged the resulting inference times. The complete graph dataset represents one probabilistic complete graph where each pair of nodes is connected by an edge. In both datasets, every node has a probability of 0.5 of being connected to another node if the abducible representing the edge is selected. Thus, the number of abducibles is the same as the number of edges. The goal is the existence of a path between nodes 1 and N, where N is the size of the graph (number of nodes). In the case of a complete graph, the number of edges, and thus abducibles, is $(N \cdot (N - 1))/2$. For the experiments with ICs, we removed paths of length two up to five: if `path(A,B,L)` is the predicate that represents the path between nodes A and B with length L, this can be imposed with `:- path(0,49,L), L < 6`.

All the main features of these five datasets are reported in Table 9.4, where #p is the number of probabilistic rules, #h the number of atoms in the head

[3]https://networkx.github.io/

per clause, #b the number of atoms in the body, #a the number of abducibles, #IC the number of ICs, #bIC and the number of atoms in the body of ICs per IC, all parametric in the size n of the program. We reported values only for the datasets with ICs, since the values for the datasets without ICs are equal except for the number of ICs that is obviously 0.

Dataset	#p	#h	#b	#a	#IC	#bIC
blood	27 + n	{3,4}	{2,3}	2	2	2
gh	2n	[1,n]	1	n	1	$\binom{n}{2} + 2$
gnb	$n \cdot (n-1)/2 + 1$	1	[1,n]	n	1	n
graph	$2(n-50) + 96$	1	1	$2(n-50) + 96$	6	1
complete graph	$n \cdot (n-1)/2$	1	1	$n * (n-1)/2$	3	1

Table 9.4: Details of the datasets.

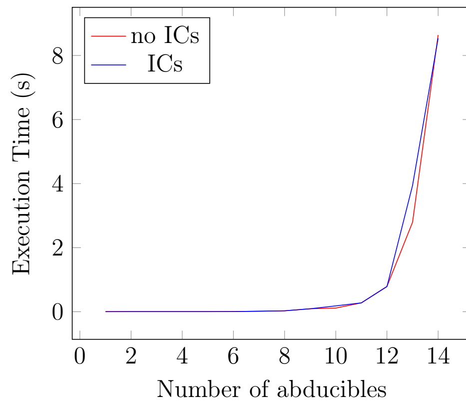
Discussion of the Results

For the dataset gh, inference times are shown in Figure 9.6a. Inference takes less than one second for programs with up to 12 abducibles for instances without ICs and up to 11 abducibles for instances with ICs. After that, in both cases, the execution time exponentially increases.

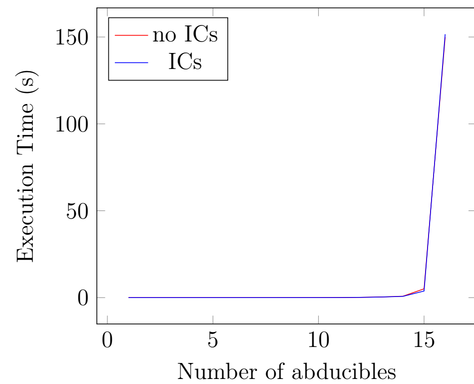
Execution times for gnb are shown in Figure 9.6b. Until 14 abducibles, execution time takes less than one second. From that number onwards, it increases exponentially. For both gh and gnb, experiments with ICs are slower than the ones without, even if for gnb the results are comparable. Similarly for the blood dataset (Figure 9.7). In the case of the dataset of size 36, the execution time exceeds 1 hour, both with and without ICs.

In the graph dataset (Figure 9.8a), the execution time increases as the number of abducibles increases, reaching exponential growth. In the case of complete graph (Figure 9.8b), the execution times for graphs of sizes up to 6 is negligible. For size 7, the required time is approximately 18 seconds (with IC) and 46 seconds (without ICs). Starting from size 8, it exceeds 8 hours. Here, experiments with ICs are faster than the ones without.

Overall, the experiments with and without ICs have comparable execution times. In case of complete graph, experiments with ICs are faster: this probably happens because constraints remove some paths in the BDD.



(a) Results for the gh dataset.



(b) Results for the gnb dataset.

Figure 9.6: Inference time as a function of the number of abducibles for the gh and gnb datasets, with and without integrity constraints.

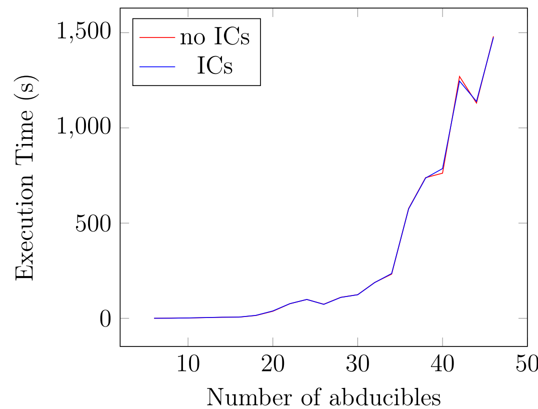
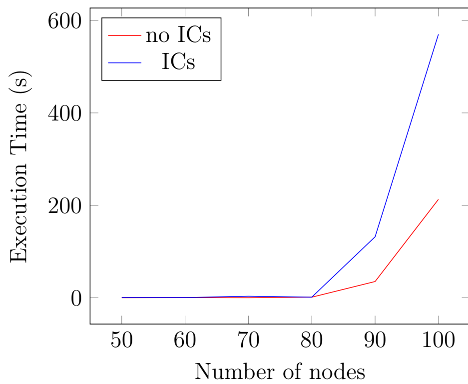


Figure 9.7: Inference time as a function of the number of abducibles for the blood dataset, with and without integrity constraints.

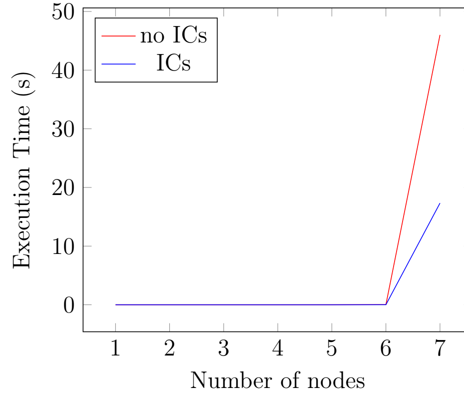
Clearly, scalability is an issue, and the execution time exponentially increases for larger instances. This is unavoidable given the complexity of the task and the expressivity of the language.

9.2 Related Work

Abduction embeds the implicit assumption that many possible explanations exist and raises the issue about which one should be selected. Adopting a purely logical setting, one may prefer the minimal ones. However, different minimal but incomparable explanations are possible (there is no total ordering



(a) Results for the graph dataset.



(b) Results for the complete graph dataset.

Figure 9.8: Inference time as a function of the number of abducibles for the graph and complete graph datasets, with and without integrity constraints.

on them). Alternatively, explanations may be selected on the basis of their reliability, so that non-minimal explanations are not discarded by default. Reliability is directly connected with (un)certainty, so there can be a connection with the domain of probabilistic reasoning.

Some works explicitly address probabilistic abductive reasoning: in [41], the authors propose a probabilistic approach to rank explanations and investigate the role of integrity constraints when performing inference. They consider a probability distribution over the truth values of each (ground) abducible, while here we set probabilities on integrity constraints.

Some proposals embed the Expectation Maximization (EM) algorithm. The PRISM [147] system does not provide support for integrity constraints, but it includes a variety of top-level predicates which can generate abductive explanations. By introducing a probability distribution over abducibles, it selects the best explanation using a generalized Viterbi algorithm and it can also learn probabilities from training data. In essence, it computes the Viterbi proof (see Section 9.2.1). For a detailed comparison of our approach with MAP and Viterbi proofs, see Section 9.2.1.

The authors of [80] introduced an abductive inference software that exploits an EM algorithm working on BDDs to evaluate hypotheses obtained from the process of hypothesis generation. It works as follows: initially, all the minimal explanations are generated. Then, they apply the EM algorithm to a formula

involving the disjunction of the hypotheses and the conjunction of the ground instances of the background knowledge, compactly represented as a BDD, to compute its probability. As the final step, the probability of each hypothesis is computed as the product of the probabilities of the literals appearing in it.

Other solutions approached abduction from a deductive reasoning perspective. For example, the one proposed in [93] exploits Markov Logic Networks (MLN) [132]. MLNs only provide deductive inference, so abduction is carried out by adding reverse implications for each rule in the knowledge base. However, this increases the size and complexity of the model, and its computational requirements. As MLNs, most statistical relational formalisms use deduction for logical inference, and so, they cannot be used effectively for abductive reasoning. The authors of [8] adopt Stochastic Logic Programs [116], considering a number of *possible worlds*. Abduction is performed by reversing the deductive flow of proofs and collecting the probabilities associated with the involved clauses. Compared to our proposal, programs are restricted to SLP, and integrity constraints are not considered. Furthermore, an implementation is currently not available.

The solution presented in [48] describes an original approach to Probabilistic Abductive Logic Programming based on Constraint Handling Rules, that allows interaction with external constraint solvers. As for our approach, it can return minimal explanations with their probabilities. They provide an implementation returning all the solutions and one returning only the most probable one. Differently from our approach, their system attaches probabilities only to abducibles, and has limitations in the use of negation, since it must be simulated by normal predicate symbols (e.g., *not_p(X)* for $\neg p(X)$). Overall, the expressivity of the constraints is more limited than in our proposal.

In the context of Action-probabilistic logic programs (ap-programs) used for modelling behaviours of entities, in [155] the authors focused on the problem of maximizing the probability that an entity takes a (combination of) action(s), subject to some constraints. This problem is called Probabilistic Logic Abduction Problem, or PLAP. Specifically, they consider the Basic PLAP setting, where the goal is fixed (a predicate checking reachability of a desired situation from the current situation) and the answer is binary. Differently from our approach, in PLAP the program is ground, and variables and constraints only

concern probabilities. Another approach that uses ap-programs for abductive query answering can be found in [156].

Some proposals approached probabilistic reasoning in abduction but did not make the ALP components probabilistic. In [126], programs contain non-probabilistic definite clauses and probabilities are attached to abducible atoms. So, there are no structured constraints, and no integrated logic-based abductive proof procedure. cProbLog [63] extends regular ProbLog logic programs, where facts in the program can be associated with probabilities, to consider integrity constraints. It comes with a formal semantics and computational procedures, resulting in a powerful framework that encompasses the advantages of both PLP (ProbLog) and Markov Logic Networks. Differently from our proposal, constraints are sharp, and thus all worlds that do not satisfy the constraints are ignored.

The discussion in [86] only considers ICs in the form of (universally quantified) *denials*, i.e., negations of conjunctions of literals. Other abductive frameworks proposed different kinds of integrity constraints: IFF [66] and its extensions, CIFF [162] and SCIFF [3] are based on integrity constraints that are clauses (i.e., implications with conjunctive premises and disjunctive conclusions). Building on these representations, the solution presented in [1] deals with probabilistic integrity constraints and proposes an associated distribution semantics. However, it considers only theories made up of constraints.

In [137] the authors propose an algorithm to learn probabilistic constraint logic theories (PCLT, a probabilistic extensions of integrity constraints [1]) from interpretations. They focus on the tasks of parameter learning and structure learning, while we focus on inference. Moreover, the target languages are different: we consider probabilistic logic programs while they consider PCLT. Similarly, in [33] the authors provide a sound and complete proof procedure to perform inference in ALP programs with probabilistic constraints and an implementation based on CHR. Here, we focus on probabilistic logic programs extended with abducibles and constraints, and we propose an algorithm working on BDDs.

A recent proposal [62] extends traditional ALP by allowing several types of integrity constraints inspired by logic operators and attaching probabilities to all components in the program (logic clauses, abducibles, and integrity con-

straints). Differently from this work, it allows ranking candidate explanations by likelihood but does not compute their exact probability.

While not explicitly computing with abduction, other systems may have a relationship to our work in that they merge logic programs, constraints, and probabilities. Specifically, Answer Set Programming (ASP) [39] may express denials and choice rules. There is a stream of work on probabilistic extensions of ASP that can deal with abduction through choice rules. Usually these works propose specific systems, implementations, or optimizations.

P-log [26] extends ASP by adding random attributes (random variables) of the form $a(X)$ where probabilistic information (understood as a measure of the degree of an agent's belief) about possible values of a is given through so-called *pr-atoms*. The logical part of a program represents knowledge which determines the possible worlds of the program, while pr-atoms determine the probabilities of these worlds. LPMLN [103] extends ASPs by allowing weighted rules based on the Markov Logic weighting scheme. LPMLN programs can be turned into P-log programs to use its reasoning engine. As to the former, the translation of non-ground LPMLN programs yields unsafe ASPs. As to the latter, the straightforward implementation of a translation of an LPMLN program into an equivalent MLN results in effective computation. PrASP [120] is a probabilistic inductive logic programming (PILP) language and an uncertainty reasoning and statistical relational machine learning software, based on ASP. It includes limited support for inference with probabilistic normal logic programs under non-ASP-based semantics.

9.2.1 Relation with MAP, MPE, and Viterbi

The tasks of Maximum a Posteriori (MAP) and Most Probable Explanation (MPE) [32], and Viterbi proof [125, 149, 154] require the selection of a subset of facts to maximizes the probability. However, there are some differences with the probabilistic abductive problem. Starting from a joint probability distribution over a set of random variables X, a set of values for a set of variables $X_e \subset X$ (evidence), and another set of variables $X_q \subset X$, $X_q \cap X_e = \emptyset$ (query variables), the MAP problem consists in finding the most probable values of the set X_q of the query variables given the evidence about X_e. In the case the set of the query variables is the complement of the set of evidence

variables with respect to the set X, the problem is called MPE.

If we consider an LPAD T, a conjunction of ground atoms e representing the evidence, and a set of query random variables X_q associated with some ground rules of T, the MAP problem requires finding an assignment x_q to variables in X_q such that $P(x_q \mid e)$ is maximized. In a formula:

$$\arg\max_{x_q} P(x_q \mid e).$$

If X_q includes all the random variables associated with all ground clauses of T, the problem is MPE. Both MAP and MPE differ from the probabilistic abductive problem, since in the latter the goal is to find a set that maximizes the probability of the query variables, rather than finding the values of the query variables. In other words, in the probabilistic abductive problem, the optimal subset of variables is unknown, while in MAP/MPE this set is known, and we need to find the associated values. Moreover, integrity constraints are not allowed neither in MAP/MPE nor in Viterbi proof tasks (discussed below). Let us clarify these differences with some examples.

Example 25. *Given the program T of Example 24 where the two last facts are made probabilistic (reported here for clarity):*

```
1 eruption:0.6;earthquake:0.3 :- sudden_er,
     fault_rupture(X).
2 sudden_er:0.7.
3 fault_rupture(southwest_northeast):0.5.
4 fault_rupture(east_west):0.4.
```

and evidence ev is `eruption`*, if all the random variables associated with all ground clauses are query variables, the MPE task finds*[4] *the most probable explanation for ev (the one with the highest probability) corresponding to the assignment x_q:*

```
[rule(1,eruption,(sudden_er,fault_rupture(southwest_northeast))),
rule(1,eruption,(sudden_er,fault_rupture(east_west))),
rule(2,sudden_er,true),
rule(3,fault_rupture(southwest_northeast),true),
```

[4]This example can be tested at `https://cplint.eu/e/eruption_mpe.pl`.

```
rule(4,null,true)]
```

Facts for predicate `rule/3` *specify respectively the clause number, the selected head, and the clause body with the selected grounding.* $P(x_q \mid ev) = 0.6 \cdot 0.6 \cdot 0.7 \cdot 0.5 \cdot (1 - 0.4) = 0.0756$.

Example 26. *Given the program of Example 25 and the evidence ev* `eruption`*, if the query variables are only those associated with lines 3 and 4, the MAP assignment*[5] x_q *is:*

```
[rule(3,fault_rupture(southwest_northeast),true),
rule(4,null,true)]
```

with probability $P(x_q \mid ev) = 0.126$*, computed as* $\frac{P(x_q,ev)}{P(ev)}$*, where* x_q *is the composite choice* $\kappa = \{(C_3, X/southwest_northeast, 1), (C_4, \{\}, 2)\}$*, and* C_3 *and* C_4 *are the third and fourth lines of the program.*

The Viterbi proof is the most probable proof for a query, i.e., it is a partial assignment (a partial possible world) such that for all assignments extending the proof the query is still true. Consider this example:

Example 27. *Given the program of Example 25, the covering set of explanations for query* `eruption` *is* $K = \{\kappa_1, \kappa_2\}$ *with*

$$\kappa_1 = \{(C_1, \{X/southwest_northeast\}, 1), (C_2, \emptyset, 1)\}$$
$$\kappa_2 = \{(C_1, \{X/east_west\}, 1), (C_2, \emptyset, 1)\}$$

κ_1 *corresponds to the partial assignment*

```
[rule(1,eruption,(sudden_er,fault_rupture(southwest_northeast))),
rule(2,sudden_er,true),
rule(3,fault_rupture(southwest_northeast),true)]
```

having probability $0.6 \cdot 0.7 \cdot 0.5 = 0.21$ *while* κ_2 *corresponds to the partial assignment*

```
[rule(1,eruption,(sudden_er,fault_rupture(east_west))),
rule(2,sudden_er,true),
rule(4,fault_rupture(east_west),true)]
```

[5] This example can be tested at `https://cplint.eu/e/eruption_map.pl`.

having probability $0.6 \cdot 0.7 \cdot 0.4 = 0.168$. *Since the Viterbi proof is the most likely explanation in the set* K, *it corresponds to* κ_1[6].

9.3 Conclusions

In this section, we extended the PITA system to perform reasoning on probabilistic abductive logic programs. Given an LPAD, a set of abducible facts, and a set of (possibly probabilistic) integrity constraints, the goal is to find the probabilistic abductive explanation that maximize the joint probability of the query and the constraints. We developed an algorithm to solve this task and tested it on several instances. The code was integrated in a web application available at `https://cplint.eu` [2].

[6]This example can be tested at `https://cplint.eu/e/eruption_vit.pl`.

Chapter 10

Integrating Constraints and Probability

The integration between logic, probability, and constraints has not received yet much attention. In this chapter, we propose two new classes of probabilistic logic programs: probabilistic *optimizable* logic programs (Section 10.1), where the probabilities of some facts can be tuned to optimize constraints involving probabilities of atoms, and probabilistic *reducible* logic programs (Section 10.2), where some facts can be removed from the theory to satisfy an objective function subject to constraints. Both these new classes of programs are a novel contribution, introduced respectively in [13] and [14]. Sections 10.3 and 10.4 conclude this chapter presenting related work and some final considerations on this topic.

10.1 Probabilistic Optimizable Logic Programs

Real-world domains are intrinsically uncertain. Just to name a few: in social networks, we are uncertain whether two people (or groups, or communities) know each other, in power networks (and integrated circuits), we are unsure about the reliability of the involved electrical components, and in road networks we are uncertain on the distribution of traffic. This partial unpredictability can be modelled with random variables. Some situations may require considering constraints: in social or collaboration networks we may want to conduct a successful advertising campaign, to obtain a target probability

to reach one specific person; in power networks we would like to minimize the effect of a system fault by optimally placing some components. Again, in road networks, we may want to maximize the probability to reach a certain destination without encountering road construction sites or traffic jams. Clearly, to solve these problems, there is the need to integrate constraints and probability. There can be other, more theoretical, domains that can benefit from this integration, such as Markov networks, where we may want to optimize the chances to reach a particular state by setting the probabilities of intermediate transitions, or probabilistic context free grammars, where we may be interested in finding the probabilities of the involved terms such that the parsing of a sentence does not change.

We choose to extend PLP since it offers a powerful and rich language where these situations can be easily represented. Given a probabilistic logic program, we address the problem of tuning the probabilities of some probabilistic facts to optimize an objective function subject to constraints on probabilities of facts and queries. To solve this, we extend the PITA reasoner [139] to allow the definition of *optimizable* facts with tunable probabilities and integrate it with an optimization solver to manage the optimization task. In such a way, we can retain the full LPAD expressive power, and we do not need to write from scratch a specialized system.

Following the ProbLog syntax, *optimizable* facts are denoted with

$$\text{optimizable}\ [\Pi_{lb}, \Pi_{ub}] :: a.$$

where a is a logical atom, and Π_{lb} and Π_{ub} are lower and upper probabilities ($\Pi_{lb} < \Pi_{ub}$) for a. Intuitively, the probability of optimizable facts can be set in the specified range to optimize an objective function subject to constraints. If the range is not specified, we set it in our implementation to $[0.001, 0.999]$. Both the objective function and the constraints can be linear or nonlinear combinations of the probability of (optimizable) facts. As usual in PLP, probabilistic facts, and so optimizable facts, are considered independent. We now formally introduce a new class of programs to manage optimizable facts and the task they aim to solve.

Definition 31 (Probabilistic Optimizable Logic Program (POLP))**.** *Given an*

LPAD $\mathcal{L}$, a set of optimizable facts $\mathcal{O}$, an objective function $\mathcal{F}$, and a set of constraints $\mathcal{C}$, the tuple $(\mathcal{L}, \mathcal{O}, \mathcal{F}, \mathcal{C})$ identifies a probabilistic optimizable logic program (POLP).

Definition 32 (Probabilistic Optimizable Problem)**.** *Given a probabilistic optimizable logic program $(\mathcal{L}, \mathcal{O}, \mathcal{F}, \mathcal{C})$, and a conjunction of ground atoms, the query (q), the probabilistic optimizable problem consists of two steps:*

- *Find a probability assignment $\mathcal{A}^*$ to optimizable facts $o_i \in \mathcal{O}$ such that the objective function is minimized (or maximized) and constraints are not violated, i.e.:*
$$\mathcal{A}^* = \underset{\mathcal{A},\ \text{subject to}\ \mathcal{C}}{\arg\min} (\mathcal{F} \mid \mathcal{A}).$$
- *Compute the probability of the query given these assignments*
$$P(q \mid \mathcal{A}^*).$$

This task can also be considered as parameter learning under constraints.

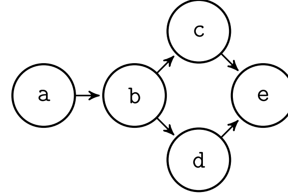
Consider the following motivating example.

Example 28. *Suppose you need to route a signal through a path composed of intermittent edges (connections), from a source to a given destination. Some edges have a fixed probability to be active, while some other edges can be controlled, and their probabilities can be set. However, the probability to reach the destination must be above a certain threshold. Furthermore, the probability of the edges that can be controlled should be set at the minimal value to reach the target probability, since setting higher probabilities also requires higher manufacturing costs. Moreover, the probabilities of the edges should be similar (i.e., their difference should be below a threshold).*

This scenario can be represented with a POLP as follows. Consider a graph with five vertices named `a`, `b`, `c`, `d`, and `e`, connected through `edge/2` facts (depicted in Figure 10.1b). Suppose that edges between `a` and `b`, `c` and `e`, and `d` and `e` cannot be controlled and have a fixed probability of 0.9, 0.3 and 0.8 respectively. Edges between `b` and `c` and `b` and `d` can be controlled and must have probability in the range $[0.3, 0.8]$. A path between two nodes is

```
1  0.9::edge(a,b).
2  optimizable [0.3,0.8]::
      edge(b,c).
3  optimizable [0.3,0.8]::
      edge(b,d).
4  0.3::edge(c,e).
5  0.8::edge(d,e).
6
7  path(X,X).
8  path(X,Y):- path(X,Z),
      edge(Z,Y).
```

(a) A possible encoding for the motivating example.



(b) Graph of the motivating example.

Figure 10.1: Program for the motivating example, together with the network graph.

represented by predicate `path/2`. The POLP shown in Figure 10.1a represents the described situation.

In Example 28, the goal is to minimize the sum of the probabilities of optimizable facts (`edge(b,c)` and `edge(b,d)`) given that the probability to reach `e` from `a` (`path(a,e)`) must be above a certain threshold. We set this threshold to 0.6. Moreover, the difference between the probabilities of the two optimizable facts should be less than a constant that we set to 0.1. To simplify the notation, in the remaining part of this section we remove the explicit probability signature from facts involved in the optimization task. In other words, if we write, for example, `edge(a,b) > 0` in a constraint, we mean that the probability of `edge(a,b)` should be greater than 0. Following Definition 31, we get:

- $\mathcal{O} = \{edge(b,c), edge(b,d)\}$
- $\mathcal{F} = edge(b,d) + edge(b,c)$
- $\mathcal{C} = \{path(a,d) > 0.6, edge(b,d) \in [0.3, 0.8], edge(b,c) \in [0.3, 0.8],$ $edge(b,c) - edge(b,d) < 0.1, edge(b,d) - edge(b,c) < 0.1\}$

Implicitly, $path(a,d) \in [0,1]$. Note that the two last expressions of $\mathcal{C}$ are an expanded version of $|edge(b,c) - edge(b,d)| < 0.1$ (absolute value).

To solve the probabilistic optimizable problem, we introduce the new predicate

```
prob_optimize/4
```

that receives as input the query (in our example `path(a,e)`), the objective function to be minimized (`edge(b,c) + edge(b,d)`), and a list of constraints. As output, it returns the probability of the query and the optimal probability assignment to optimizable facts. So, for Example 28 with the thresholds chosen before, the query would be

```
prob_optimize(
    path(a,e),
    [edge(b,c) + edge(b,d)],
    [path(a,e) > 0.6, edge(b,c) - edge(b,d) < 0.1,
        edge(b,d) - edge(b,c) < 0.1],
    Assignments).
```

Lower and upper bounds are directly introduced by the facts, without the need to be specified also in the constraints list. The main idea is that a POLP induces a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, where n is equal to the number of optimizable facts. In Figure 10.1a, $n = 2$.

As discussed in Section 7.1, probabilistic logic programs are converted into an alternative form through knowledge compilation. Here, we also convert a POLP into an alternative form. We choose BDDs, since they are already used by the PITA reasoner [139]. The result of the compilation of a POLP into a BDD for Example 28 and query `path(a,e)` (denoted with p_{ae}) is shown in Figure 10.2, where e_{xy} stands for `edge(x,y)`. We can extract an equation from a BDD by following all the paths, multiplying the nodes encountered during the traversal and adding all the partial equations of every path. For example, in Figure 10.2 there are three paths that go to 1 with an even number of complemented edges. The resulting function is the sum of the product of the probabilities of the edges of these three paths. If we consider the leftmost path that goes to one with a regular arc we get the following function:

$$f(e_{bc}, e_{bd}, e_{ab}, e_{de}) = (1 - e_{bc}) \cdot e_{bd} \cdot e_{ab} \cdot e_{de}.$$

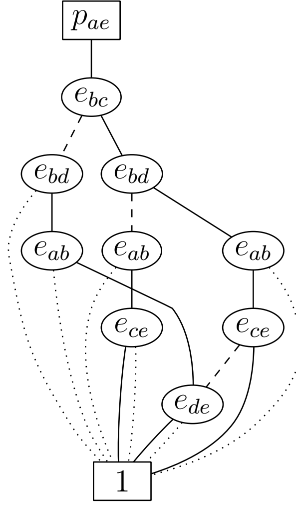


Figure 10.2: BDD for the program shown in Figure 10.1, where a dashed line represents a 0-edge, a solid line the 1-edge, and a dotted line the 0-complemented edge.

Algorithm 7 Function OPTIMIZEPROB: optimization of probability of random variables.

1: **function** OPTIMIZEPROB(query,objective,constraintsList,algorithm)
2: root ← Compute the BDD for the query
3: paths ← PATHSPROB(root)
4: query equation ← convert paths into a symbolic equation
5: Simplify query equation
6: Replace constraints from constraintsList involving the query with query equation
7: assignments ← Call the nonlinear optimization solver with [objective, constraintsList, algorithm]
8: prob ← evaluate(query equation | assignments)
9: **return** [assignments, prob]
10: **end function**

In the case of PLP, nodes of the BDD are associated with a probability value. Here, we keep symbolically the nodes representing optimizable facts (i.e., with their name) while substituting nodes representing probabilistic facts with the associated probability. So, the previous function becomes

$$f(e_{bc}, e_{bd}) = (1 - e_{bc}) \cdot e_{bd} \cdot 0.72$$

where $0.72 = e_{ab} \cdot e_{de}$. In this way, we reduce both the number of variables and arithmetic operations.

To solve the probabilistic optimizable problem, we start by constructing the BDD for the query. Then, we extract the equation from the BDD with Algorithm 8. It goes as follows: first, the BDD is reordered to have nodes

Algorithm 8 Function PATHSPROB: computation of all the paths of a BDD and of their probability.

```
1: function PATHSPROB(root)
2:     root′ ← REORDER(root)                                          ▷ BDD reordering
3:     TablePaths ← ∅
4:     TableProb ← ∅
5:     if root′.comp then
6:         comp ← true
7:     else
8:         comp ← false
9:     end if
10:    return PATHSPROBREC(root′, comp, TablePaths, TableProb)
11: end function
12: function PATHSPROBREC(node, comp, TablePaths, TableProb)
13:    comp ← node.comp ⊕ comp
14:    if var(node) is not associated to an optimizable fact then
15:        p ←PROB(node, TableProb)                                     ▷ Call to PROB
16:        if comp then
17:            Res ← [1 − p, []]
18:        else
19:            Res ← [p, []]
20:        end if
21:    else
22:        if TablePaths(node.index) ≠ ∅ then
23:            return TablePaths(node.index)
24:        else
25:            Lp0 ← PATHSPROBREC(child0(node), comp, TablePaths, TableProb)
26:            Lp1 ← PATHSPROBREC(child1(node), comp, TablePaths, TableProb)
27:            Res ← []
28:            for all path ∈ Lp0 do
29:                if path.prob > 0 then
30:                    Res ← Res ∪ {path ∪ [node.index, 0]}
31:                end if
32:            end for
33:            for all path ∈ Lp1 do
34:                if path.prob > 0 then
35:                    Res ← Res ∪ {path ∪ [node.index, 1]}
36:                end if
37:            end for
38:        end if
39:        Add node.index → Res to TablePaths
40:    end if
41:    return Res
42: end function
```

corresponding to optimizable facts first in the order (on top, next to the root). Then, the function PATHSPROBREC is recursively called until a node associated with a probabilistic fact (or the terminal node) is encountered. From there, the function PROB from [60] (Algorithm 1) is called. This returns an empty paths list and the computed probability. After that, in the nodes associated with optimizable facts, the paths at the 0 and 1 children are extended with the current node index (provided that the obtained path has a probability greater than 0). Two tables are used to store already computed probability values and paths, to speed up the process and avoid performing multiple times

the same operation. Once the list of all the paths where optimizable variables are kept with their name is computed, we can extract the equation representing the query, that we called *query equation*, as described before: we multiply together the nodes and the probability for each path, and then add up the results. Consider a node n: if it appears selected along a path, we multiply the current result by n. If it is not, by one minus n. Clearly, nodes not appearing in a path from source to the 1 terminal node does not appear as well in its equation. For example, for Figure 10.2, the resulting list of paths obtained using Algorithm 8 is $[[0.774, [[e_{bd}, 1], [e_{bc}, 1]]], [0.27, [[e_{bd}, 0], [e_{bc}, 1]]], [0.72, [[e_{bd}, 1], [e_{bc}, 0]]]]$, where $0.774 = e_{ab} \cdot (e_{ce} + (1 - e_{ce}) \cdot e_{de})$, $0.72 = e_{ab} \cdot e_{de}$, and $0.27 = e_{ab} \cdot e_{ce}$ (computed by following the correspondent paths on the BDD), which represents the equation $e_{bd} \cdot e_{bc} \cdot 0.774 + (1 - e_{bd}) \cdot e_{bc} \cdot 0.27 + e_{bd} \cdot (1 - e_{bc}) \cdot 0.72$.

The equation obtained from Algorithm 8 is simplified and substituted in the constraint(s) involving the query. Then, it is passed, together with all the user-specified constraints, to the (nonlinear) optimization solver. Finally, once the optimal values are computed (provided that the problem has a solution), the probability of the query is computed by evaluating the query equation, where symbolical variables are substituted with their optimal values. For the program shown in Figure 10.1a, with the previously discussed constraints, one possible solution is given by $f(0.6352, 0.7352) = 1.3704$, with a probability for the query `path(a,e)` equal to 0.6. The whole process is reported in Algorithm 7.

We now motivate two crucial steps of Algorithm 7: the reordering of the BDD and the simplification of the query equation. The reordering of the BDD is fundamental, since it allows to directly apply the function PROB (Algorithm 1) once we reach a node not associated with an optimizable fact (as already discussed in Section 9.1.2). In this way, we obtain a compact expression, where combinations of the probability of random variables are multiplied by a single numerical value. The simplification of the query equation is also important for increasing its compactness and reducing the number of performed operations. To see this, consider again the equation extracted from the BDD of Figure 10.2, reported here for clarity: $e_{bd} \cdot e_{bc} \cdot 0.774 + (1 - e_{bd}) \cdot e_{bc} \cdot 0.27 + e_{bd} \cdot (1 - e_{bc}) \cdot 0.72$. We have 10 operations (multiplications and summations) to perform. If we simplify it, we obtain $-0.216 \cdot e_{bc} \cdot e_{bd} + 0.27 \cdot e_{bc} + 0.72 \cdot e_{bd}$: now the number of operations is 6. This equation is called multiple times during the

solution of the optimization problem, so the time spent to simplify it (which is often negligible, as we will show later in the experiments) is amortized. Furthermore, this process may also reduce the impact of the BDD structure: the order of the variables in a BDD determines its size, and there are several BDD that represent the same equation, more or less compactly. If we directly use the extracted equation, this may be long and may involve more computations than necessary.

To study the complexity of the task, we need to consider that answering probabilistic queries is #P-complete in general (see Section 7.1). The construction of the BDD is as well #P-complete, so the probabilistic optimizable problem is at least in that class. The reordering of the BDD can be performed polynomially in its size (Section 7.1.1). We decided to perform only one reordering of the BDD (to move the optimizable variables) and then simplifying the extracted equation. An alternative approach consists of reordering the BDD until a compact equation is found. This is clearly infeasible since the possible orderings of variables are exponential in number.

10.1.1 Experiments

We implemented[1] Algorithm 7 using C, leveraging some existing libraries: NLopt [85] to solve the optimization problem, CUDD [157] for the operations on BDDs, and the function `simplify` from the Python SymPy package [111] to simplify the query equation. The simplification is guided by some heuristics, and iteratively tries to apply some possible simplifications, even if, in general, there is no guarantee that the equation with the minimal size is found. Finally, we used SWI-Prolog [171] version 8.3.15 for the logic part.

To test our algorithm on real-world scenarios, we selected several datasets from [143] with graph structure representing social networks, collaboration networks, road connections, and power networks. We pre-processed the data to convert them into a set of `edge(a,b)` facts (eventually adding a probability value or prepending the functor `optimizable` and an associated probability range) representing a connection between node `a` and `b`. For all the experiments, the goal is to constrain the probability of the paths between a random

[1] Source code available at: `https://bitbucket.org/machinelearningunife/polp_experiments`

source (`Source`) and a random destination (`Destination`) to be greater than 0.8 (provided that the path exists), while minimizing the sum of the probabilities of optimizable `edge/2` facts. Notice that some of these may not be involved in the path from a source to a destination, but it is difficult to spot them without running the query. We set 50% of the total nodes to be optimizable with a range $[0.001, 0.999]$. The remaining are probabilistic facts with probability 0.5. Since we randomly choose source and destination, results are averages of 10 runs. The query was `path(Source,Dest)`, and we constrain its probability to be greater than 0.8 with `path(Source,Dest) > 0.8`, where `path/2` is the predicate shown in Figure 10.1a.

For a second set of experiments, we generate complete graphs of increasing size. As before, the goal is to constrain the probability of a path between node index 1 and n, where n is the size of the graph, to be greater than 0.8. The distribution of optimizable and probabilistic facts is the same as before (50-50). Differently from the previous programs, here the number of nodes is substantially smaller. However, the solution of the optimizable problem is harder, since the graph is fully connected, and this reflects an increasing number of paths from the root of the BDD to the terminal node, and thus the extracted equation is long and complex.

For all datasets, we tested three local gradient-based optimization algorithms available in NLopt [85]. Two are based on conservative convex separable approximations [158], denoted with MMA and CCSAQ, and one is based on sequential quadratic programming [100], denoted with SLSQP. We conducted the experiments on the same machine described in Section 7.2.1. For NLopt, we set the tolerance to 10^{-5}, i.e., the optimization process stops when the variation of the objective function between two consecutive evaluations is less than this value. Execution times are limited to 8 hours and are computed with the SWI-Prolog predicate `statistics/2` with keyword `walltime`.

Results are shown in Table 10.1 and Table 10.2, which report the average execution time (BDD generation plus simplification and optimization) and the average value of the objective function (sum of the probabilities of the optimizable facts) for all three algorithms. For the real-world graphs, we also tabled the standard deviations of the values of the objective function and the number of vertices and edges for each dataset. For this experiment, the

best results are marked in bold. In Table 10.1, the dataset bio stands for bio-DM-LC, ca for ca-netscience, E60 for ENZYMES_g60, IIP for internet-industry-partnerships, p494 for power-494-bus, p662 for power-662-bus, rtf for reptilia-tortoise-fi-2008, rc for road-chesapeake, rt for rt-retweet, soc for soc-tribes, and web for webkb-wisc (all from [143]).

	Features		Time (s)			Objective Value			StdDev (Obj)		
Dataset	|V|	|E|	C	M	S	C	M	S	C	M	S
bio	658	1129	2595	4934	**1072**	171	115	**2**	146	115	**1**
ca	379	914	2859	2137	**387**	47	70	**2**	103	103	**1**
DD244	291	882	2070	2355	**521**	76	76	**2**	103	103	**1**
E60	10	36	**25**	30	37	**0.057**	**0.057**	**0.057**	**0.114**	**0.114**	**0.114**
IIP	219	613	1079	690	**209**	65	127	3	81	66	**1**
p494	494	586	3026	3898	**1052**	151	122	**2**	142	142	**1**
p662	662	906	16167	7990	**2292**	109	286	**2**	183	182	**1**
rtf	283	418	271	211	**48**	36	59	**1**	52	54	**1**
rc	39	170	47	35	**7**	5	18	**1**	13	22	**1**
rt	97	117	21	13	**3**	7	17	**1**	13	15	**1**
soc	16	58	133	133	**11**	**0.114**	**0.114**	**0.114**	**0.164**	**0.164**	**0.164**
web	265	530	829	712	**146**	38	50	**2**	62	66	**1**

Table 10.1: Results for the network experiments. C, M and S stand respectively for CCSAQ, MMA, and SLSQP algorithms. |V| is the number of vertices and |E| the number of edges respectively.

N	C (s)	M (s)	S (s)	C (Obj)	M (Obj)	S (Obj)
3	1.6	1.7	0.4	1.548	1.548	1.548
4	8.2	2.7	0.4	1.725	0.735	0.735
5	4.5	4.7	0.9	1.391	1.391	1.459
6	10.7	12.4	1.3	0.715	0.715	0.715
7	202.4	193.8	30.1	0.83	0.83	0.83
8	1,360.3	1,600.6	177.6	0.83	0.83	0.83

Table 10.2: Results for the complete graphs experiments.

We decided to keep together the execution times for BDD construction, query equation extraction and simplification, and optimization since the first three are negligible with respect to the optimization time. However, the simplification of the equation can be a little more expensive than its extraction from the BDD, even if it is still in the order of seconds.

Overall, the best algorithm is often SLSQP, with a significant difference in both terms of execution time and value of the objective function. This is evident also from the experiments on complete graphs, where it is almost 10

times faster than CCSAQ and MMA for larger instances, but with comparable values for the objective function. For the complete graph of size 9 the execution time exceeds 8 hours since the nodes have a high degree. Consequently, the number of paths exponentially increases, and the length of the query equation explodes. The possible connections are clearly crucial for the execution time. For example, the dataset bio-DM-LC has approximately 200 vertices and edges more than the dataset power-494-bus, but the two execution times are comparable, especially for SLSQP. This also occurs for other networks we tested, still from [143], such as soc-firm-hi-tech, ia-crime-moreno, lp_adlittle, and soc-wiki-vote, where some queries do not terminate within 8 hours.

As expected, the bottleneck of this approach is the solution of the optimizable problem. The other operations have, in practice, less or no influence on the execution time. To solve this, we can, for example, reduce the tolerance of the algorithm (but at the cost of more imprecise results), leverage techniques from lifted inference, or adopt representations alternative to BDDs.

10.2 Probabilistic Reducible Logic Programs

While in POLP the goal is to set the probabilities of some facts, there can be situations where we may want to completely remove facts from the theory. To solve this, we introduce a new class of programs that we called probabilistic *reducible* logic programs, where some facts can be marked as reducible with the meaning that these can be removed from the program itself. The goal is to remove as many reducible facts as possible, while maintaining the validity of some constraints involving random variable values. In this case, differently from POLP, the goal can be considered as structure learning since we want to select a subset of the possible facts and not set their probabilities.

We now formally introduce these types of programs and the associated task. Without loss of generality, we suppose that constraints are of the form `eq > 0`, where `eq` is a (possibly) nonlinear equation involving probabilities of atoms.

Reducible facts are denoted with the special functor `reducible` and have the following syntax:

$$\text{reducible } \Pi :: a.$$

where a is a logical term with associated probability $\Pi \in]0, 1]$. For uniformity with LPADs, the syntax reducible $a : \Pi$ is also allowed.

Definition 33 (Probabilistic Reducible Logic Program)**.** *Given an LPAD $\mathcal{L}$, a non-empty set of reducible facts $\mathcal{R}$, and a non-empty set of constraints $\mathcal{C}$, the tuple $(\mathcal{L}, \mathcal{R}, \mathcal{C})$ identifies a probabilistic reducible logic program (PRLP).*

Definition 34 (Probabilistic Reducible Problem)**.** *Given a probabilistic reducible logic program $(\mathcal{L}, \mathcal{R}, \mathcal{C})$, the probabilistic reducible problem consists in finding the minimal subset of reducible facts to keep such that the constraints in $\mathcal{C}$ are satisfied. In formulas:*

$$R^* = \underset{R \subseteq \mathcal{R},\ \texttt{subject to}\ \mathcal{C}}{\arg\min} |R|.$$

This task involves both discrete variables (reducible facts) and nonlinear constraints, so it can be classified as a mixed-integer nonlinear programming (MINLP) problem.

The next example motivates the definition of this new class of programs.

Example 29 (Motivating Example - Viral Marketing, adapted from [164])**.** *We target a set of people (forming a social network) to advertise a new product, but there is uncertainty on the possible connections. On a higher level, nodes can represent communities (or groups of people linked by the same interests). Some time after the beginning of the campaign, due to an economic crisis, the number of targeted people must be reduced as much as possible, and we need to choose to stop the campaign towards some of them. The higher is the number of target people, the higher are the costs. However, we want to maintain a lower bound on the probability that one or more of them still buy the product. In general, a person can buy a product if she/he is directly targeted or if some trusted friend buys the product. This scenario can be encoded by the following program.*

```
1 reducible 0.9::target(a).
2 reducible 0.2::target(b).
3 reducible 0.6::target(c).
4 reducible 0.7::target(d).
5
```

```
 6 knows(X,Y) :- friend(X,Y).
 7 knows(X,Y) :- friend(Y,X).
 8
 9 0.8::friend(a,b).
10 0.7::friend(b,d).
11 0.6::friend(a,c).
12 0.5::friend(c,d).
13
14 buys(X):- target(X).
15 buys(X):- knows(X,Y), buys(Y).
```

The probabilistic facts `friend(A,B)` *represent that* `A` *is friend with* `B` *with a certain probability, and the predicate* `knows/2` *states that* `A` *knows* `B` *if* `A` *is friend with* `B` *or* `B` *is friend with* `A`*. The reducible facts* `target/1` *represent the targeting of a person, and they may be removed. These have an associated probability since the targeting action may fail for some reason. Note that this value indicates the probability that the fact is true if it is not removed (as happens for normal probabilistic facts), not the probability that it will be removed. Finally, the predicate* `buys(X)` *states that* `X` *buys the product if* `target(X)` *is true or both* `knows(X,Y)` *and* `buys(Y)` *are true.*

For example, with all the four reducible facts included, the probability of the query `buys(d)` *is 0.920. Suppose that we want to reduce the number of targeted people while keeping the probability of the query above a certain threshold (set to 0.9). There are four possible facts to remove and* 2^4 *possible combinations. The optimal combination of people to target is given by*

```
{target(a),target(c),target(d)}
```

(`target(b)` *is removed from the program) and the probability of* `buys(d)` *becomes 0.9131.*

As already noticed, there are several similarities between POLP and PRLP. Also here, to solve the probabilistic reducible problem, we extend the PITA reasoner [139] and introduce a new predicate called `prob_reduce/4` with the signature

```
prob_reduce(AtomList,ConstraintsList,Algorithm,Result)
```

where `AtomList` is a list of atoms involved in the constraints, `ConstraintsList` is a list containing one or more (possibly nonlinear) constraints, and `Algorithm` is the selected algorithm (exact or approximate). All these three are input variables. The computed result (the set of selected reducible facts) is unified with the variable `Result`. As before, we remove the explicit probability signature: the name of an atom involved in constraints stands for its probability. If we consider the program shown in Example 29 and we want to maintain the probability of `buys(d)` above 0.9, we will call `prob_reduce/4` as:

```
prob_reduce(
    [buys(d)],
    [buys(d) - 0.9 > 0],
    exact,
    Result
).
```

The `exact` keyword selects the GEKKO solver [31] for the optimization problem. Alternatively, we implemented an approximate greedy algorithm that can be selected with the keyword `approximate`: it iteratively removes the reducible fact that provides the least difference in probability for all constraints when removed.

The whole pipeline is like the one for POLP, and is shown in Algorithm 9. First, we extract all the equations for the terms in `AtomList` by computing the BDDs and traversing them. The results are stored in a list. Since there can also be normal probabilistic facts in the program, the BDD is reordered to move them at the bottom. In this way, we can apply the function PROB (Algorithm 1) and obtain a more compact equation, as discussed in Section 10.1. After that, the terms in `constraintList` are replaced with the corresponding equations (line 10) and the selected solver is called. The exact solver leverages GEKKO [31], while the approximate works as follows: at each iteration, we compute the difference between the left part of the constraint (i.e., the one before `> 0`) with and without every reducible fact that has not been already removed (line 22 function COMPUTEDIFFERENCE). Then, we try to remove the constraint that gives the least reduction (line 29). If this violates one of the constraints, the iteration stops. Otherwise, its probability is set to 0 and these steps repeat until no more removals are possible. For the approximate

algorithm, in case the removals of two facts give the same probability value, the first according to the order of appearance is selected. For the exact algorithm, the choice is managed by the solver itself.

If we consider the query `buys(d)` of Example 29, with all four reducible facts included it has a probability of 0.92. At the first iteration of the approximate algorithm, we obtain from the function ComputeDifference the following values for the four reducible facts: $[0.0747, 0.0069, 0.0232, 0.1866]$. Then, the function RemoveOne is called: the second variable gives the least reduction while keeping the constraint true ($0.920 - 0.0069 - 0.9 > 0$), so it is removed by setting its probability to 0. At the next iteration, the computed values are $[0.0928, -, 0.0262, 0.2027]$ ($-$ is a placeholder to denote a variable that has already been removed), and the probability of the query is 0.9131. None of these variables can be removed, since we would get respectively -0.07978, -0.0131, and -0.1896, all being less than 0. So, the solution `{target(a),target(c),target(d)}` is returned. For this example, the approximate algorithm also computes the optimal solution. However, in general, there are no guarantees on the optimality. The discussion about the complexity of Algorithm 9 is like the one provided for Algorithm 7.

10.2.1 Experiments

We implemented[2] Algorithm 9 using C for the construction of the BDDs, Python for the integration and implementation of the solvers, and Prolog (SWI-Prolog version 8.3.15) for the logic programming part. To test the effectiveness of our proposal, we conducted several experiments on the same machine used in sections 7.2 and 10.1.1. We set the maximum execution time to 8 hours and the maximum memory usage to 8GB. We used the GEKKO APOPT[3] solver with the following options: `minlp_maximum_iterations = 100000`, `minlp_branch_method = 2`, `minlp_as_nlp = 0`, `minlp_integer_tol = 0.00005`, `minlp_gap_tol = 0.00001`, `nlp_maximum_iterations = 5000`, and `minlp_max_iter_with_int_sol = 5000`. Execution times are `real time` values obtained using the bash command `time`.

[2]The implementation and the datasets are available at: `https://bitbucket.org/machinelearningunife/prlp_experiments`

[3]`https://gekko.readthedocs.io/en/latest/global.html`

Algorithm 9 Function MINIMIZEREDUCIBLES: minimizing the number of reducible facts.

```
1: function MINIMIZEREDUCIBLES(atomList,constraintsList,algorithm)
2:     equationsList ← []
3:     for all atom ∈ atomList do
4:         bdd ← COMPUTEBDD(atom)
5:         reorderedBdd ← REORDER(bdd)
6:         pathsList ← COMPUTEALLPATHS(reorderedBdd)
7:         symbolicEquation ← CONVERTINTOSYMBOLICEQUATION(pathsList)
8:         equationsList ← equationsList ∪ [symbolicEquation]
9:     end for
10:    list ← REPLACEWITHSYMBOLICEQUATION(constraintsList,equationsList)
11:    if algorithm is exact then
12:        assignments ← SOLVEEXACT(constraintList)          ▷ Compute exact solution
13:    else                                                  ▷ Approximate algorithm is used
14:        factsList ← [atom,true] for all atoms in atomList
15:        endOpt ← false
16:        while endOpt is false do
17:            gl ← []
18:            for all constraint in constraintsList do
19:                g ← []
20:                for all [fact,selected] in factsList do
21:                    if selected is true then
22:                        g ← g ∪ COMPUTEDIFFERENCE(fact,factsList,constraint)
23:                    else
24:                        g ← g ∪ {-1}
25:                    end if
26:                end for
27:                gl ← gl ∪ g                   ▷ Add the current list of facts to the total list
28:            end for
29:            index ← REMOVEONE(gl,constraintsList)
30:            if index == -1 then
31:                endOpt ← true                           ▷ No more variables can be removed
32:            else
33:                factsList[index].selected = false
34:            end if
35:        end while
36:        assignments ← factsList
37:    end if
38:    return assignments
39: end function
```

We tested our algorithm on datasets having a graph structure. The edges are represented with `friend/2` facts represent knowledge relationships. From these, we created programs with the following structure:

```
1 buys(X):- target(X).
2 buys(X):- knows(X,Y), buys(Y).
```

The structure of the predicate `knows/2` distinguishes two versions of the datasets: an easy one (directed graph) and a hard one (undirected graph). For the former, the predicate `knows/2` has the following structure (directed graph)

```
1 knows(X,Y):- friend(X,Y).
```

while for the latter, it has an additional clause (undirected graph)

```
1 knows(X,Y):- friend(Y,X).
```

In both cases, every `target/1` fact is reducible and has an associated probability of 0.5. Overall, all the programs have this structure, but with a different number of reducible and probabilistic facts.

To test the performance of our approach when programs have an increasing number of groundings, we generated complete graphs (abbreviated with KN) up to size 15. By default, we used the exact solver and fallback to the approximate one if a solution cannot be computed (this is often the case, since the obtained equations are long and complex, except for smaller instances). Results are shown in figures 10.3 and 10.4. The first one shows the execution times for query `buys(1) - 0.5 > 0` for both directed and undirected instances. The exact solver is feasible only for graph sizes up to 8 since, for larger graphs, the solver returns an error caused by the excessive length of the equation. The accuracies are shown in Figure 10.4, computed as the difference between the lowest acceptable value of the probability and the computed value. For example, if the constraint is `buys(1) - 0.5 > 0` and the computed value for `buys(1)` is 0.6, the gap is $0.6 - 0.5 = 0.1$. We selected values starting from 0.5 (as in this example) up to 0.9 with a step of 0.1 for the threshold probability. For the directed KN graph with value 0.9 and size 5 the gap is negative, meaning that, even without removing any fact, the probability cannot be greater than this threshold.

As expected, after a certain size (9 for undirected and 15 for directed), the execution time explodes, due to an increasing number of groundings. The gap is about 0.05 for all the values except for 0.6, where it is approximately 0.15 for both directed and undirected. This may be due to the structure of the graph that does not allow a combination of facts such that the probability can be so low.

As a second experiment, we used the probabilistic graph dataset taken from [32], consisting of a set of 10 graphs with a number of edges ranging between 50 and 500 with step 50. Overall, there are 10 different graphs for every size, since their generation is not deterministic (they are generated using a Barabási-Albert model where the number of edges to attach from a new node to existing nodes was set to 2). All the edges are reducible facts with an associated probability of 0.9, even if some of them may not be involved in

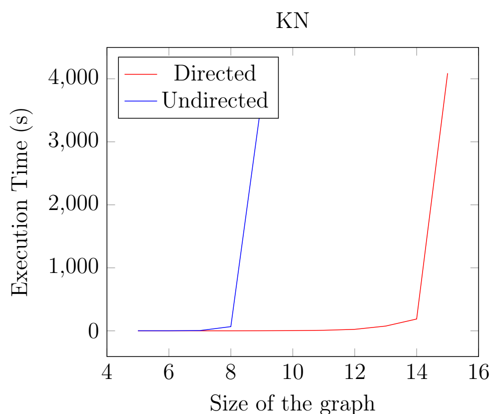
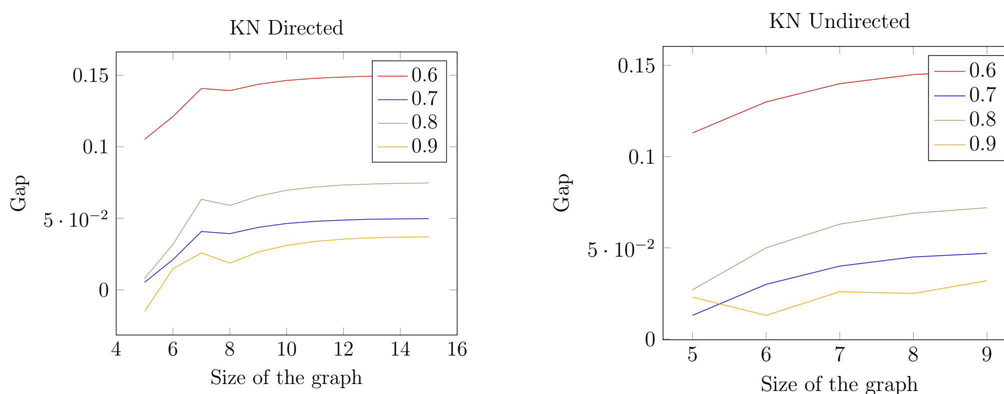


Figure 10.3: Execution time for directed and undirected complete graphs.



(a) Results for directed graphs.

(b) Results for undirected graphs.

Figure 10.4: Computed gaps of the approximate algorithm on both directed and undirected complete graphs.

the query. The goal is to constrain the probability of the path between node of index 0 and index size of the graph - 1 to be greater than 0.5. Results of the approximate algorithm are shown in Table 10.3, where a dash indicates that a solution cannot be computed given the specified time and memory constraints. Overall, the execution time increases as the size of the graph increases. However, the variance increases as well, since nodes can be more or less connected, possibly generating very complex structures.

Finally, to test in detail the exact solver, we selected the datasets ca-netscience, power-494-bus, rt-retweet, and webkb-wisc from [143]. As before, these programs consist of directed graphs, and the goal is to constrain the probability to reach a random destination from a random source to be greater

Dataset	50	100	150	200	250	300	350	400	450	500
1	1.88	1.979	802.418	3523.055	3.121	1111.115	171.09	-	11.668	115.06
2	1.77	30.293	2.459	1249.843	7.834	4.052	6.264	7.179	301.37	-
3	4.78	115.845	3.58	14533.699	540.74	9.714	98.1	3.939	5.995	8265.052
4	1.8	2	3.823	190.697	-	103.688	30.582	192.336	145.644	16.393
5	26.33	1.756	1422.327	-	92.451	1692.265	-	9241.692	-	-
6	1.669	2.027	1.981	4308.51	174.596	423.971	28.414	250.672	-	-
7	2.7	16.654	1456.541	4.068	3.959	4.897	4.098	100.503	61.22	11.634
8	1.768	492.654	251.157	149.31	73.821	43.264	4689.642	21.82	18	17.5
9	1.705	1.772	2.131	2.431	459.622	57.278	11.003	78.667	10.95	136.639
10	2.852	2.624	88.738	3.39	6.841	330.987	419.668	92.207	-	627.594
Mean	4.72	66.76	403.51	2662.78	151.44	378.12	606.54	1109.89	79.26	1312.83
Variance	58.55	23642.07	359016.56	22501362.06	42731.49	330434.32	2362332.53	9305915.23	12070.10	9445459.51

Table 10.3: Execution time for the probabilistic graph dataset of [32].

Dataset	Nodes	Edges (total)	Execution Time (s)
ca-netscience	379	914	3.80
power-494-bus	494	586	1.94
rt-retweet	97	117	1.83
webkb-wisc	265	530	7.17

Table 10.4: Results for the exact algorithm on graphs from [143].

than 0.1. Both probabilistic and reducible facts (that are equally split) have an associated probability of 0.9. The results shown in Table 10.4 are averages of 10 runs, all with the same settings but different sources and destinations. Since, for these datasets, the ratio between the number of edges and the number of nodes is small, the resulting BDD for the query is small, as evident from the execution time (seconds).

Overall, the exact algorithm can be applied only for small domains (or with a compact BDD representation), since the complexity of the equation rapidly increases. In general, the approximate algorithm seems to provide a good trade-off between accuracy and execution time.

10.3 Related Work

Work on parameter learning is related to POLP. In parameter learning, the task is to learn the probabilities of probabilistic facts given a set of positive and negative examples. However, explicit constraints on probability of probabilistic facts are usually not considered. Here, we learn the parameters of the program (probabilities) starting from a set of constraints, and so there is no need for a training set, since our algorithm solves a constrained optimization problem.

For parameter learning, one of the first approaches was proposed in [146], where the goal is to find the maximum likelihood parameters of special atoms, called *msw* atoms, not present in the dataset. The authors propose a naive implementation of the EM algorithm, later improved in [150]. EM is also used in [34] to learn the parameters of LPADs represented as BDDs. Another technique is LFI-ProbLog [72] where parameters of ProbLog programs are learning from partial interpretations. LFI-ProbLog is included in ProbLog2 [64].

Gradient-based methods for parameter learning are used in [71], where derivatives are computed directly on the BDD representation of the program. Here, we also use gradient-based methods, but we do not compute gradients on the BDD. Rather, we use the equation obtained from the BDD (query equation), after simplification, in all the computations. In this way, we traverse the BDD only once. A related idea can be found in [96], where the authors present aProbLog, an extension of the programming language ProbLog [60], that can be used to solve different tasks. One of these is sensitivity analysis, i.e., seeing how a change in the probability of the facts affects the probability of a query. However, they cite the task in passing, and they do not consider further constraints on the probability of the variables. The same problem is discussed in [122]. A similar line of research is represented by DTProblog [164], where probabilistic logic programs are extended with Boolean decision variables associated with a cost. The goal is to maximize an expected utility by selecting an optimal subset of these variables. Also in this case, no constraints are considered.

Another related solution is the one presented in [102], where the authors introduced an algorithm to combine PLP and Constraint Programming to solve decision-theoretic tasks. As in that paper, we use a compact representation of the probabilistic logic program (they use SDD, we use BDD), and extend an already existing tool (they extend ProbLog, we extend PITA). However, they restrict the type of constraints involved (linear constraints over sum of Boolean decision variables), and they do not consider the computation of optimal probability values.

Stochastic constraint programming (SCP) [170] is a technique that combines probability and constraints. SCP programs are composed of decision variables that can be set and stochastic variables that follow a probability dis-

tribution. The goal is to find a subset of decision variables such that constraints are satisfied with a certain probability. Here, we do not consider probabilistic constraints (we have hard constraints that must be always true) and we search for optimal probability values for optimizable facts that are present in the program (they cannot be removed, as decision variables) but with a tunable probability. Furthermore, these approaches are often tailored to solve specific problems such as games [5], scheduling [106], or sequential planning [24].

An idea related to PRLP can be found in [58], where the authors propose an algorithm to find the smallest (with less than k clauses) ProbLog program that maximizes the likelihood of a training set. The algorithm removes one clause at the time, starting from the one that yields the highest likelihood when removed. Differently from this work, we use a set of constraints to guide the search, instead of a training set, and we do not restrict a priori the number of clauses (in our case, facts). Our target is to find the minimal set of facts, while their goal is to maximize the likelihood.

PRLP are also different from (probabilistic) abductive logic programs [12]: in abduction, the goal is to find a subset of facts (often minimal) that explains a query, but usually constraints between probability values are not considered. Similar differences can be found with the MAP and MPE task [32], where the goal is to find the most probable value of a set of variables given evidence on other variables, and with k-best [95] and Viterbi task, where the goal is to find the best k explanations for a query (in Viterbi, k is set to 1).

10.4 Conclusions

In this chapter, we introduced two new classes of probabilistic logic programs: probabilistic optimizable logic programs [13] and probabilistic reducible logic programs [14]. The tasks solvable with these new proposals are similar to parameter and structure learning respectively. For both, we considered constraints on random variable probabilities and then we defined a constrained optimization problem. We introduced several motivating examples and conducted numerous experiments to test the performance of the proposed algorithms on real-world scenarios. Overall, these two classes widen the tasks solvable with PLP and represent a step towards the integration between prob-

ability and constraints.

Part IV

Applications of Probabilistic Logic Programming

Chapter 11

Blockchain

Initially proposed in 1990 [74] as a method to securely timestamping digital documents, blockchain technology has been mass adopted after Satoshi Nakamoto published the paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [117] (even if the term *blockchain* was never used, as a unique word, in it). In this chapter, we introduce the basic concepts of blockchain systems, focusing in particular on Bitcoin and Ethereum.

11.1 Structure

A blockchain is a distributed ledger shared among the users. This ledger is composed of several blocks which are collections of transactions. All the blocks, except for the first one called *genesis block*, have a single ancestor. Transactions in a block are ordered and linked together by cryptography functions, an idea that goes back to 1990 [74]. Blocks are composed of a header and a body: the header contains some information to check the consistency of the transactions stored in the body. One of the first, and currently most famous, blockchain is the Bitcoin blockchain [117], proposed in 2008, that manages more than three hundred thousand transactions per day[1].

Usually, a blockchain has an associated currency, called cryptocurrency, used to perform transactions and interact with the system. For all blockchain types, users (and non users) can download a full copy and validate transac-

[1] `https://www.blockchain.com/it/charts/n-transactions`

tions themselves, since the blockchain is public[2]. The validation of blocks and transactions follows different steps, according to the technology adopted in the considered blockchain. One of the main drawbacks of blockchains is that, due to the huge number of transactions issued every day, tracking them requires significant computational power and huge storage.

The three properties that every blockchain should have are: scalability, decentralization, and security. However, as the blockchain trilemma states, only two of these three can hold at the same time.

We now focus on the Bitcoin blockchain and illustrate some of its features.

11.1.1 Bitcoin

In Bitcoin, users are identified by an address (composed of a public key and a private key) and can send transactions to other users. Usually, transactions consist of a movement of bitcoin from a user to another user. A transaction is composed of a set of input addresses and a set of output addresses. The value sent as input is often greater than the output value: the difference is collected as a *transaction fee* by the user (miner) that decides to include it in a block. Users are called *peers*, and some of them are *miners*. To append a block to the blockchain, a miner needs to provide a Proof of Work (PoW, *hashcash* in case of Bitcoin). PoW is an algorithm belonging to the family of consensus algorithms and consists in solving a hard computational problem, namely finding a hash for a block such that its value is less than a certain predefined target called *difficulty*. The difficulty value is dynamically adjusted to keep the number of discovered blocks constant over time (approximately one block every ten minutes). The PoW can, however, be easily verified so that every user can check the validity of the proposed block.

The varying difficulty is the main bottleneck that limits the scalability of the system. To speed up computations, usually miners group themselves into mining pools[3], to aggregate their computational power (also called hash rate) and to increase the probability of finding a valid hash, since providing

[2]Here, we do not consider private blockchains or other hybrid solutions that hide part of the data.

[3]In this dissertation, we will always use the term miner to indicate a mining pool, unless otherwise specified.

a solution of the PoW without sharing resources is almost impossible. This aggregation, however, increases the centralization, and goes against the core decentralization principle of this blockchain. Moreover, the formation of mining pools increases the probability of attacks [142]. Despite these drawbacks, the bitcoin blockchain is currently completely mined by pools[4].

The whole blockchain is stored in a peer-to-peer way, and factors such as network latency can generate forks, i.e., a bifurcation in the flow of the main chain. However, only the longest chain is the one considered valid. In case of two chains of the same length, the next block that will be discovered will have only one of the two as ancestor, so it will identify the main chain. The presence of several alternative chains could be an indicator of a double spending attack [22, 124, 142], a situation where a user wants to spend the same amount of currency multiple times.

Once a user issues a transaction, this goes into the mempool, the set of unconfirmed transactions. Miners select some of them to be included into a block and starts the PoW. A constraint in Bitcoin imposes that the size of a block must be less than 1Mb (even if there are some possible variations that we will discuss later). If a miner can solve the problem, its block will be added to the blockchain, and it receives the fees present in the included transactions plus a reward (through a *coinbase* transaction) in bitcoin. Transactions included in blocks are considered confirmed. The block reward decreases as the number of blocks increases, to avoid the generation of an infinite number of bitcoin. Users can attach high fees (usually measured in satoshi per byte, where 1 satoshi = 10^{-8} bitcoin) to prioritize transactions, since miners would gain more profits by including them. However, there must be a balance between fees and priority: if fees grow too quickly, users may be unwilling to pay such a high amount for a transaction, and consequently they will abandon the system.

Different elements complicate the computation of the optimal amount of fees for a transaction. Suppose there are four transactions waiting to be included in a block: A with size 150kb and fee 150, B with size 250kb and fee 300, C with size 350kb and fee 450, and D with size 450kb and fee 600. These values are greater than the average transaction size, but they are used only to illustrate the process. We can compute the ratio fee/size (a quantity called *fee*

[4] `https://btc.com/stats/pool`

rate) for all the four and obtain 1 for A, 1.2 for B, 1.28 for C, and 1.3 for D. If we sort the transactions in descending order of fee rate, we get D, C, B, and A. A miner could select the ones with the highest associated fee rate to maximize the profits. However, in this case, transactions D, C, and B cannot be stored in the same block since their total size (450 + 350 + 250 = 1050kb) is greater than the block limit (1000kb = 1Mb). Consequently, the miner needs to solve a knapsack problem to select the best transactions, a well-known NP-complete problem.

Consider now instead a situation where there are different dependent transactions. For example, suppose that C depends on A (it spends an output of A). In this case, to get the reward from C, the miner should include both A and C into the block, even if A has the lowest fee rate of the pool of available transactions. This scenario is called "Child Pay for Parent", where a child transaction (C) with a higher fee rate helps the parent transaction (A) with a lower fee rate, by spending one of its outputs.

Moreover, the computation of the optimal fee rate is complicated by miners mining empty blocks, to avoid wasting time checking the validity of the last block and selecting a set of transactions. Also forks complicate the fee estimation, since the invalidated transactions can be considered again. Clearly, block discovery time and number of transactions issued per second influence the computation.

There are several applications to compute the optimal fee rate. One of them is available in Bitcoin Core[5] (a Bitcoin client) and is accessible using the command `estimatesmartfee`. The output of the command is the optimal fee rate to attach to a transaction to have it confirmed with high probability within N blocks, where N is selected by the user and can be up to 1008 (at the moment of writing). In a nutshell, the algorithm[6] works as follows: transactions are grouped into exponentially spaced buckets. Each bucket boundary is 1.05 times greater than the previous one. The computation of the optimal value is based on the number of transactions that enter in each bucket and the number of transactions included in a block within the target. The process is further refined and gives more importance to recent blocks than older ones.

[5] `https://bitcoin.org/en/bitcoin-core/`

[6] Code available at `https://github.com/bitcoin/bitcoin/blob/master/src/policy/fees.h`

Despite all the difficulties arising when considering fees, these cannot be removed: removing fees from transactions will allow some malicious users to send an infinite amount of transactions (since they are free), completely blocking the system.

Another feature that limits bitcoin scalability is the maximum size of a block, fixed to 1Mb. This value is at the heart of a very controversial topic[7]: an increase of the size will allow the system to process a larger number of transactions, but at the cost of higher computational requirements to store and manage the blockchain, reducing its decentralization. Moreover, forks will be more likely to happen, due to a slower block propagation time. Finally, a change in the block size will require a hard fork (a drastic change), since it would be backward incompatible, increasing the probability of system failure if some nodes do not update the protocol, and reducing the reliability of the overall system.

During the years, several proposals (called Bitcoin Improvement Proposals, BIPs) tried to increase the number of Bitcoin transactions that can be managed per second. Segregated Witnesses (SegWit) was one of the first, proposed at the end of 2015[8]. This solution suggested to increase the capacity of a block by removing signature data from a transaction, and by introducing the concepts of Virtual Size and block weight, measured in weight units instead of bytes.

Currently signatures are needed for executing transactions. If a user wants to send transactions from multiple addresses to one, each of these require its own signature, increasing the size of the transaction and making it more expensive. A current proposal that will solve this problem is the introduction of Schnorr signatures [109, 152]: after their implementation, users controlling multiple addresses that want to gather funds from them to spend in a single transaction will need only one signature, making the transaction lighter.

Schnorr signatures combined with SegWit will increase the Bitcoin scalability, but the system will still be limited. There are alternative solutions under rapid development that aim to construct a new layer on top of a blockchain, without modifying the structure of the blockchain itself. Lightning Network (LN) [129] is one of them, and we discuss it in Section 11.1.3.

---

[7] `https://en.bitcoin.it/wiki/Block_size_limit_controversy`

[8] `https://en.bitcoin.it/wiki/Bitcoin_Improvement_Proposals`

At the moment of writing, all blockchains are affected by some problems. PoW-based blockchains, such as Bitcoin [117] and Ethereum [173] are secure and (theoretically) decentralized, but not very scalable [55], since their consensus algorithm requires solving a hard computational problem. Currently, Bitcoin can handle approximately 7 transactions per second, while Ethereum 15. This is clearly a limitation and prevents their adoption as every-day payment methods. Alternative consensus algorithms such as Proof of Stake (PoS), Delegated Proof of Stake (DPoS) are more scalable but less decentralized. There is a plethora of possible consensus algorithms, see [44] for an overview and a detailed comparison.

Here, we surveyed some of the basic concepts associated with Bitcoin needed to understand the experiments that will be discussed in the next sections. We only scratched the surface of Bitcoin: for an in-depth and comprehensive treatment see [4].

11.1.2 Smart Contracts

Several blockchain systems allow running code in them using *smart contracts*, initially proposed in 1994 [160] as computer protocols to facilitate a self-enforcing agreement between two parties. Essentially, smart contracts are programs written in a quasi-Turing[9] complete programming language that runs in a blockchain environment. In the case of Ethereum, the main language is Solidity, but there are some alternatives. Bitcoin also has a language for smart contracts called Bitcoin script, but it is intentionally not Turing-complete.

The key feature is that users can enforce contracts without the need of a central authority and without reciprocal trust. The execution of a smart contract is deterministic, so it always produces the same result when provided with the same inputs, even if there are some solutions to provide, for example, randomization. Moreover, in most of the blockchain systems, smart contracts cannot be modified, so an error in the code cannot be easily fixed. For this reason, several tools have been proposed to analyse them to guarantee (at least in principle) the desired behaviour; these will be discussed in Section 12.1.

[9]The programming language itself is Turing complete, but the execution of a smart contract requires the usage of a certain amount of currency. Once the associated currency is terminated, the execution stops, and hence the quasi-Turing completeness.

11.1.3 Lightning Network

There are several proposals to increase the number of processed transactions, such as sharding [108] and sidechains [25]. For Bitcoin, one of the currently most promising solutions is the Lightning Network [129] (LN). In the following, when we use the term Lightning Network, we consider the Bitcoin Lightning Network, even if the concept of LN can be extended to other blockchains.

The goal of LN is to build a layer of nodes on top of an underlying blockchain, where transactions can be sent and processed in a faster way. In these systems, users can open a bidirectional payment channel through a transaction on the main chain. The opening transaction locks some funds on the channel that can be used for small and fast payments without interacting with the main blockchain. This allows users to avoid paying high transaction fees and waiting long confirmation time. For security and privacy reasons, the balance of a channel and its funds distribution at the two ends are unknown[10]. The state of a channel (its balance) can be updated with a commitment transaction not published on the main chain. To close the channel, the two involved parties must agree on its balance and then publish a closing transaction. However, in the case of uncooperative parties, Hashed Timelock Contracts (HTLCs)[11] ensure that funds are not lost. Moreover, payments can also be routed towards nodes not directly connected, provided that there exists a path between them, using multi-hop payments. Again, HTLCs ensure the security of these, in case, for example, an intermediate node refuses to forward the payment.

When the source and the destination of a payment coincide, the routing operation is called *rebalance*. This operation involves a circular payment, so it is useful when a node wants to refill a channel.

Payments are usually associated with small fees for intermediate nodes. These can be of two types: fee base and fee rate. The former is fixed while the latter is proportional to the size of the routed payment. Both are (usually) requested from a node that forwards a payment. The capacity of a channel is expressed in satoshi (10^{-8} bitcoin), the fee base in thousandths of a satoshi,

[10]https://github.com/lightningnetwork/lightning-rfc/blob/master/07-routing-gossip.md

[11]https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts

and the fee rate in millionths of a satoshi.

Chapter 12

Analysis of Blockchain-related Scenarios

In this chapter, we illustrate how several blockchain-related scenarios can be modelled using Probabilistic Logic Programming. Sections 12.1, 12.2, and 12.3 discuss respectively how we can leverage PLP to model smart contracts, a centralization of the hashing power with a subsequent double spending attack, and transaction fees. Finally, Section 12.4 introduces both a deterministic and a probabilistic model of the (Bitcoin) Lightning Network. The content of this chapter is novel and was introduced in [10, 11, 16, 17, 18, 20, 22].

12.1 Smart Contract Analysis

In this section, we focus on smart contracts written in Solidity, a language that can be compiled into bytecode and executed on the Ethereum Virtual Machine (EVM) [40]. However, the model we propose can be extended to a general programming language. As said before, the code of a smart contract cannot be modified once deployed on a blockchain. Moreover, its execution of is deterministic. For these reasons, several tools to analyse them have been proposed during the years. For example, the authors of [107] performed static analysis on the code, while the authors of [89] used a symbolic model to check the execution flow. Another proposal is [76], where the authors defined the EVM in a language that can be understood by a theorem prover, used to ensure the correctness of the operations. All these solutions, however, can

detect bugs but cannot provide a quantitative impact of errors in the code. In [43] the authors develop a new framework to extract utility values from a smart contract through game theory considerations. Differently from them, we propose a PLP encoding of Solidity smart contracts that allows us to study how a bug quantitatively affects the execution. In this way, we can rely on the well-studied semantics of probabilistic logic programs, and we do not need a new specialized framework.

The interactions with a smart contract can be considered as probabilistic since the involved parties are not under the control of a central authority. The issuer may be interested in studying how the obtained profit evolves, while the users may be interested in the possible rewards they would get by interacting with it (for example, in the case of gambling games). We can compute these values by translating a smart contract into a probabilistic logic program, where each function corresponds to a predicate. This process involves two steps: first, the smart contract is translated into a logic program (Prolog), then probabilistic facts are added. The translation from a smart contract function to a logic predicate is straightforward: every function corresponds to a predicate with the same name and same number of arguments. Moreover, if needed in the execution flow, the members of the globally `msg` object, such as `msg.sender` or `msg.value` can be added as input arguments. The return value of the function (if present) is as well added to the arguments of the predicate.

The following code shows a simple Solidity smart contract simulating a bank.

```
1  contract simpleBank {
2      address owner;
3      mapping(address => uint) balances;
4
5      constructor() public {
6          owner = msg.sender;
7          balances[msg.sender] = 1000;
8      }
9
10     function transfer(address receiver, uint
           amt) public {
```

```
11          require(balances[msg.sender] >= amt);
12          require(msg.sender != receiver);
13          balances[msg.sender] -= amt;
14          balances[receiver] += amt;
15      }
16  }
```

The function `constructor` is executed when the contract is created: it sets the owner of the contract and issues 1000 tokens to the creator. The function `transfer` accepts two parameters: the address `receiver` of the receiver and the amount `amt` the caller wants to transfer to the receiver. First, it checks whether the sender is different from the receiver and whether it has enough funds to transfer. Both these conditions are evaluated using the function `require`, that tests the condition and, in case of failure, throws an exception that stops the program. If the conditions are satisfied, the balances of both parties are updated accordingly.

This small contract can be translated into a probabilistic logic program as follows. We simulate the storage reserved for a smart contract by introducing two more arguments in the predicate encoding a particular function that uses it, one for the input list mapping all addresses to balances, and one for the same list but updated with the new balances. We define a predicate `find/3` to retrieve the balance of the sender (identified with `Sender`) from `BalanceList` and return it into `BalanceSender`. Similarly for the balance of the receiver. Then, we perform the same checks of function `require`, and we update the balances and the list using another pre-defined predicate `update/3`. This is needed since Prolog variables cannot be modified once they are bound. We now have a smart contract written in Prolog, that we turn into a probabilistic logic program by adding some probabilistic facts. For example, we suppose that the transferred amount from a user to another user (this operation can correspond to placing a bet, where funds are sent to the address where the contract is stored) is uniformly distributed between 0.5 and 2 Ether (the currency adopted in Ethereum). This can be expressed, following the `cplint` hybrid program syntax (see Section 8.2.1) with:

```
1  amount(A):uniform(A,0.5,2.0).
```

Overall, we obtain the following code:

```
1  amount(A):uniform(A,0.5,2.0).
2  transfer(Receiver,Amt,Sender,BalanceList,
     NewBalanceList):-
3      find(Sender,BalanceList,BalanceSender),
4      find(Receiver,BalanceList,BalanceReceiver),
5      amount(Amt),
6      BalanceSender >= Amt,
7      Sender \= Receiver,
8      NewBalanceS is BalanceSender - Amt,
9      NewBalanceR is BalanceReceiver + Amt,
10     update(BalanceList,Sender,NewBalanceS,
         BalanceList1),
11     update(BalanceList1,Receiver,NewBalanceR,
         NewBalanceList).
```

We can compute the expected transferred value using the predicate

`mc_expectation/4`

provided by the MCINTYRE [134] module.

12.1.1 Experiments

To see the advantages of a probabilistic logic encoding of a smart contract, we conducted several experiments involving a smart contract for transferring tokens, one for a Ponzi scheme, and one encoding a gambling game, all taken from the Ethereum mainnet. We used the same machine described in Section 7.2.1. The execution time is computed with the built-in SWI-Prolog [171] predicate `statistics/2`, while the memory usage is the value `maxresident` obtained with GNU Time[1]. For each experiment, we used the predicate `mc_expectation/4` with 1000 samples.

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

Transfer

In the first experiment (transfer), we modelled a scenario where `N` users trade (burn or transfer) some tokens. The function `burn` and `transfer` are the ones available at the address `0xB8c77482e45F1F44dE1745F52C74426C631bDD52`. In our simulations, each user starts with 100 tokens. We want to know how many transfers are needed to obtain a situation where a single user has more than 180 tokens. Transfers are random, and the amount is uniformly distributed between 1 and 100. We also include a probability of 5% that tokens are burnt instead of traded. All these values were chosen to illustrate the overall process. Table 12.1 represents the relation among number of users, execution time of the experiments, memory usage, and expected number of transactions. As expected, the number of needed transfers to create a situation in which a user has more than 180 tokens increases with the number of users.

The Solidity transfer function described before checks whether the sender and the receiver are different. However, there are some real-world cases where this check is omitted[2], resulting in an unexpected generation of extra tokens. Our method can also be effective to spot bugs and coding errors like this, clearly if the conversion between imperative language (Solidity) and declarative (logic) language (Prolog) is correct. To confirm this, we modified the previous code by removing this check and ran again the same experiment, and we compute the total amount of circulating tokens, represented as the sum of the balances of the involved users. In every run, we choose two random (possibly the same) users and perform a transfer. Results in Table 12.2 shows that, at the end of the simulation, the total amount of circulating tokens exceeds the initial amount.

Ponzi and Pyramid Schemes

Ponzi schemes are a very common type of smart contracts, present in both Bitcoin [28, 167] and Ethereum [27]. A Ponzi scheme is a financial fraud that promises a high return of the investments, as long as the number of involved users keeps increasing. Clearly, this model quickly becomes unsustainable. In these scenarios, probabilistic models of smart contracts are fundamental to

[2] `https://gist.github.com/loiluu/0363070e1bada977f6192c8e78348438`

Table 12.1: Details for the transfer experiment without bug.

# of users	Time (s)	Memory (Mb)	Expected Value
5	1.643	52.744	192.724
25	8.717	102.924	421.516
50	23.431	155.636	661.514
75	44.172	202.428	874.234
100	71.367	246.136	1071.335
125	101.48	285.208	1249.186
150	140.116	327.488	1441.242

Table 12.2: Details for the transfer experiment with bug.

# of users	Time (s)	Memory (Mb)	Initial Amount	Final Amount
5	0.755	33.324	500	627.262
25	4.967	95.996	2500	2592.483
50	12.834	154.196	5000	5077.37
75	23.828	204.036	7500	7568.241
100	37.663	253.204	10000	10064.181
125	53.451	294.352	125000	12561.246
150	72.883	344.072	150000	15058.114

compute the expected payoff and avoid being cheated.

For this experiment (pyramid), we consider the code of a well-known pyramid scheme (a variation of a Ponzi scheme) called Rubixi, available at the address `0xe82719202e5965Cf5D9B6673B7503a3b92DE20be`. This example is usually presented to see how a vulnerability allows anyone to become the owner of the contract and withdraw the collected fees [9]. Here, we ignore this problem since we only want to conduct a quantitative analysis of the possible profit that can be obtained without exploiting this bug. The logic of the contract is simple: a user can send at least 1 Ether through the *fallback* function, a function that is executed when none of the provided functions matches the called one. When the contract receives a certain number of tokens, it adds the new participant to the participants list and redistributes the accumulated value to the others if some conditions are met. We may be interested, for example, in knowing the amount of collected fees in a certain time interval, or computing the number of participants we need to wait before receiving a payment. To answer the latter question, we model this contract with a probabilistic logic program and we suppose that the amount of token sent is uniformly distributed

Table 12.3: Details for the pyramid experiment.

# of users	Time (s)	Memory (Mb)	# of users to wait
5	0.159	13.384	13.783
50	1.997	37.232	69.814
100	6.111	59.384	111.138
150	12.681	85.768	152.868
200	22.051	113.436	194.760
300	50.852	174.420	278.393
400	92.934	225.832	361.172

between 0.9 and 2, also considering a situation where a user sends less than the minimum required amount. In this case, the amount is added to the collected fees, but the user cannot participate in the scheme. From our analysis reported in Table 12.3, we see that, for example, the fifth user entering the game must wait at least 13 more people to enter the scheme before receiving a payoff.

Gambling

According to DappRadar[3], in April 2020 half of the most used dApps (web applications with the logic implemented by a smart contract) are gambling games. For our experiment (gambling), we selected the code stored at the address `0x999999C60566e0a78DF17F71886333E1dACE0BAE` that allows to bet on multiple games such as dice, roulette, or poker, whose outcomes are computed by considering several payout masks. Randomization is obtained by combining an externally generated `commit` value provided as input for the bet with other values. We simulate a player betting on the outcome of a single die. We draw the amount of the bet from a Poisson distribution with different values for the mean, and sampled the transaction fees from a uniform distribution between 0.07 and 0.2 Finney (1 Finney = 10^{-3} Ether)[4]. Figure 12.1 and Table 12.4 show the results. Unsurprisingly, the decrease of the expected payout is related to the amount of the bet and the number of trials.

[3] `https://dappradar.com/`

[4] These values are selected by considering data from `https://bitinfocharts.com/ethereum/` (April 2020)

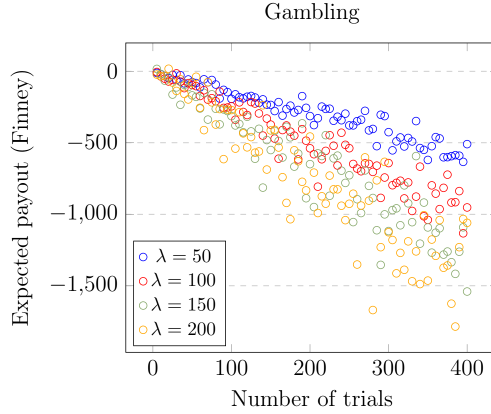


Figure 12.1: Expected payout of a consecutive number of trials. λ represents the mean of the Poisson distribution.

Table 12.4: Resource usage for the gambling experiment with $\lambda = 150$.

# of trials	Time (s)	Memory (Mb)
5	0.517	9.344
50	4.890	11.016
100	8.757	18.924
150	12.938	28.22
200	17.439	27.076
300	27.011	143.896
400	37.437	201.996

12.1.2 Conclusions

In this section, we discussed a probabilistic logic encoding of a Solidity smart contract that can be useful for testing the contract without deploying it, with the possibility of finding bugs, as shown in the experiments. Moreover, the adoption of a (probabilistic) logic language for smart contracts will allow almost everyone to write them, since a logic language is often more interpretable with respect to an imperative language such as Solidity. This idea is also supported by [50] and [79], where the authors discuss the possible implications of a logic language for smart contracts. Finally, our proposal is still far from being completed: a formal model of the translation between Solidity and PLP is missing, This will ensure that the model exactly corresponds to the real contract.

12.2 Hashing Power Centralization and Double Spending

The formation of large mining pools in the Bitcoin blockchain can eventually lead to a situation where a single entity controls more than 50% of the total computing power, as already happened in 2014[5]. In this case, a majority attack will be possible, where a group of people start mining an alternative chain with the goal to revert some transactions. This scenario will make the system completely unreliable. Here, we model the protocol proposed in [61] to limit hashing power centralization and evaluate the probability of a double spending attack [91, 124].

12.2.1 Preventing the Formation of Large Pools

To prevent the formation of large mining pools, the authors of [29, 61] proposed a two-phase PoW. The first step is the same as the current PoW. The second step consists in signing the header of a block with the private key of the address that will receive the mining reward and then in finding again a new hash that respects some constraints (less than a target value). Basically, it requires to solve twice the PoW. If we call X and Y the difficulties of the two PoWs, their ratio is crucial to prevent the formation of large pools according to the authors. However, an optimal value is not provided in the papers. This second step will require the pool operator to share its private key with all the components of the pool, allowing everyone with that key to access the stored funds and possibly steal them. Thus, the pool operator will share the key only with trusted miners, preventing the formation of a large pool.

In this new proposal, a miner can be in one of the following four states: 0) it tries to solve the original PoW. Upon success, it will move to state 1. 1) It tries to find a second hash to satisfy the second part of the protocol. 2) It has found both hashes, it receives the reward and goes back to state 1. 3) Another miner found both hashes, so the miner will stop working on the current hash and start with another one, going back to state 0. These four states can be described by the Markov chain shown in Figure 12.2, where states are indicated

[5]https://en.bitcoinwiki.org/wiki/GHash.IO

by a_0, a_1, a_2, and a_3.

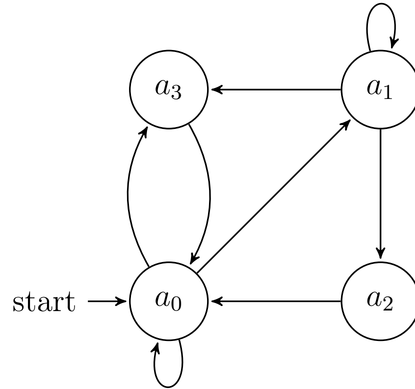


Figure 12.2: Markov chain of the model.

If we suppose both X and Y finite (in [29] and [61] X has a finite value while Y is infinite, i.e., every hash is acceptable), we can model a race between two miners `a` and `b` with the following code:

```
1 a_found_y(_):0.15.
2 b_found_y(_):0.25.
3 b_found_x(_):0.10.
4 found_y(S):- a_found_y(S); b_found_y(S).
```

Here, we suppose that `a` finds the correct Y hash with probability 0.15, and `b` with probability 0.25. `found_y/1` is true when `a` or `b` have found the correct second hash at state `S`. We can describe the transitions between states with:

```
1 trans(a0,S,a1):1.0/50; trans(a0,S,a0)
     :1.0-1.0/50:- \+b_found_x(S).
2 trans(a1,S,a2):0.15;trans(a1,S,a1):1.0-0.15:-
     \+b_found_y(S).
```

Transition between state `a0` and `a1` will happen with probability 1/50 and only if `b` has not yet found the correct first hash for the current block (in this case, there is no need to keep searching for this hash). Similarly, `a` will move from `a1` to `a2` with probability 0.15 only if `b` has not yet found the second hash. This code models the described situation:

```
1 trans(a0,S,a3):- b_found_x(S).
```

```
2 trans(a1,S,a3):- b_found_y(S).
3 trans(a2,S,a0):- found_y(S).
4 trans(a3,S,a0):- found_y(S).
```

If b finds either the first or the second hash, a will move to state a3 independently from its current state. Once a is in a3, it will move to a0 to start this process again. With the previous code we modelled the behaviour of a. The code for modelling the behaviour of b is similar, where a is replaced by b.

We can write a predicate reach/3 to simulate the change of state:

```
1 reach(S, I, T) :-
2     trans(S, I, U),
3     reach(U, next(I), T).
4 reach(S, _, S).
```

This code states that starting at state S at time I, state T is reachable if there is a transition (trans/3) from state S to state U at time I and T is reachable from U at the next time point. Finally, we can approximate the probability to reach state a2 starting from a0 by using predicate mc_sample/3 from the MCINTYRE module [134]. For example, we can sample 1000 times reach(a0,0,a2) and compute its probability with mc_sample(reach(a0,0,a2),1000,P), obtaining P = 0.068.

12.2.2 Double Spending

As previously discussed, there can be situations where multiple chains are mined simultaneously. In this case, the next block will have a crucial role, since it will identify the longest chain, and so the valid one. Because of this, users should wait at least six blocks[6] before considering the transaction completely confirmed. One of the most studied scenarios that can happen in these circumstances is the double spending attack[7], where a user spends twice the same amount of bitcoin. The process goes as follows: first, the attacker creates a transaction T1 with A as the output address and publishes it. We suppose that T1 is included into block B1. Then, he/she starts mining a private fork (chain) that does not include T1 but includes transaction T2 whose output

[6]https://en.bitcoin.it/wiki/Confirmation

[7]https://en.bitcoin.it/wiki/Irreversible_Transactions

is the address of the attacker itself and not A. Figure 12.3 depicts this initial position.

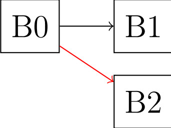


Figure 12.3: Initial state of the double spending attack. Block $B1$ with transaction $T1$ is inserted in the chain after $B0$ while the attacker starts mining another block ($B2$) without $T1$ inside and with $B0$ as ancestor.

After a certain number of blocks built on top of B1, the recipient of the transaction is convinced that T1 is valid (Figure 12.4).

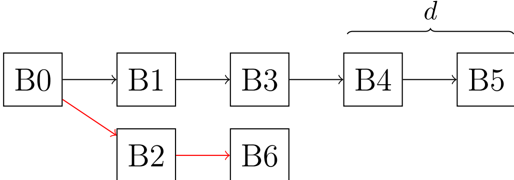


Figure 12.4: General case. The "honest" chain has built 3 confirmation blocks on top of $B1$ ($B3, B4, B5$) while only one block ($B6$) has been built on top of $B2$ by the attacker. In this figure, d represents the distance between the honest and the secret chain and is used to evaluate the advantage of the honest chain over the attacker.

Now, the attacker needs to make his/her chain longer than the honest one to be successful in the attack. In this case, the private chain will be published and considered the valid one, and all the transactions in the blocks between B1 and the last block will be invalidated (Figure 12.5).

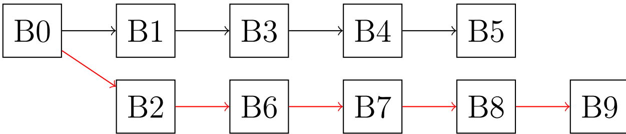


Figure 12.5: Successful attack. The attacker has built a longer chain (marked in red). The attacker will now publish all blocks from $B2$ to $B9$ and so all blocks from $B1$ to $B5$ in the black chain will not be considered valid because they are part of a chain which is not the longest one.

We make some simplifications in our models: the total hash rate of the

network as well as the difficulty are constant, and there is only one attack going on (all the miners except for the attacker mine the honest chain).

The attack can be described by two functions: the attacker's potential progress function and the catch up function. The former returns the number of blocks m mined by the attacker when the honest miners have found n blocks. In the original bitcoin paper [117], this function is defined by a Poisson distribution

$$P(m) = \frac{e^{-\lambda}\lambda^m}{m!}$$

where q is the hash rate of the attacker, $p = 1 - q$ the hash rate of the honest miners, and $\lambda = np/q$. However, in [142] the authors describe this process with a negative binomial (Pascal) distribution:

$$P(m) = \binom{m+n-1}{m} p^n q^m.$$

In both cases, even if the attacker has already found a longer chain, it cannot publish it until N confirmations are reached, otherwise the victim of the attack will be aware of it. Once the N confirmations have been created, the race between the attacker and the honest chain starts. Here, we are only interested in knowing whether the attacker will succeed, and not in how long it will take. This process, according to [117] and [142], can be modelled with a random walk. If d is the difference between the two chains (as in Figure 12.4), an increase of d by 1 means that the honest miners have found a block before the attacker. We can define the probability that the attacker will catch up from Z blocks behind [142] as

$$P(\text{catch_up} \mid Z = z) = \begin{cases} 1 & \text{if } q \geq p \\ (q/p)^z & \text{if } q < p \end{cases}$$

and express this using the following probabilistic logic program:

```
1 move(T,1):0.7; move(T,-1):0.3.
2
3 walk(InitialPosition):-
```

```
4       walk(InitialPosition,0).
5
6   walk(0,_).
7   walk(X,T0):-
8       X > 0,
9       X < 1000,
10      move(T0,Move),
11      T1 is T0+1,
12      X1 is X+Move,
13      walk(X1,T1).
```

We suppose that the distance between the chains increases by 1 with probability 0.7. If the gap between the two chains is greater than 1000, we consider the attack failed. Predicate `walk/2` encodes the random walk.

The attacker's potential progress functions proposed in [117] and [142] can be modelled in the `cplint` hybrid program syntax (Section 8.2.1) as:

```
1   attacker_progress_poisson(X):poisson(X,Lambda).
2   attacker_progress_pascal(X):pascal(X,N,P).
3
4   success_poisson:-
5       attacker_progress_poisson(A),
6       V is NumberOfConfirmations - A,
7       (   V = 0 ->
8           true;
9           walk(V)
10      ).
11
12  success_pascal:-
13      attacker_progress_pascal(A),
14      V is NumberOfConfirmations - A,
15      (   V = 0 ->
16          true;
17          walk(V)
18      ).
```

With `attacker_progress_poisson/1` and `attacker_progress_pascal/1` we sample, respectively, a value from a Poisson probability distribution with parameter `Lambda` and a value from a Pascal probability distribution with parameters `N` (number of failures) and `P` (success probability). After that, we compute the length difference `V` between the two chains (the d value from Figure 12.4). If the difference is 0, the attack is successful. Otherwise, we consider a binomial random walk with `V` as the initial position. We can use the predicate `mc_prob/2` and `mc_sample/3` again from the MCINTYRE module to get the probability values. Figure 12.6 shows the obtained results using `Lambda` $= 10 \cdot (0.3/0.7)$, `N` $= 10$, `X` $= 0.3$, and `NumberOfConfirmations` $= 10$, and these are in accordance with the values computed in [117] and [142].

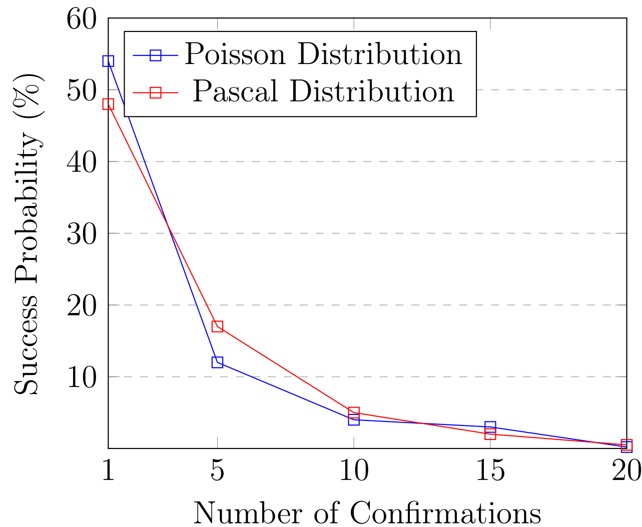


Figure 12.6: Success probability of a double spending attack by considering Poisson and Pascal probability distributions for the number of blocks mined by the attacker.

12.2.3 Conclusions

In this section, we encoded with PLP two models from [29, 61, 117] and [142], confirming their effectiveness. The first model proposes an alternative protocol to prevent the centralization of the hashing power while the second simulates a double spending attack. Several extensions are possible, for example by considering a variable hash rate or modelling alternative consensus algorithms

such as PoS or DPoS.

12.3 Transaction Fees

As introduced in the previous sections, Bitcoin transaction fees are a very complex topic: miners want to maximize the reward while users want to minimize the required fees to process a transaction, with several variables involved in the computation. In the following, we consider Bitcoin. The time required to discover a block can be represented with a Poisson distribution with parameter $\lambda = 10$ [38]. This probability distribution is commonly used to model the number of events occurring in a fixed interval of time. A prerequisite is that the events are independent and they occur at a constant rate, as it happens for block discovery: the discovery rate is kept fixed by an adjustment of the difficulty of the PoW every 2016 blocks, and each discovery is independent.

Other variables that influence the optimal fee value are the size of the transactions, the size of the blocks, and the number of transactions issued per second. We modelled the first two values with normal distributions, and the third with a Poisson distribution. A normal distribution is a continuous probability distribution, while the size of a transaction is a discrete variable that would be better modelled with a discrete distribution, such as the Poisson. However, thanks to the central limit theorem, a Poisson distribution with mean λ can be approximated with a Gaussian distribution with mean and variance λ for sufficiently large values of λ. We retrieved from `blockchain.com` (October 2019) the values of the average size of a transaction and the average number of transactions per second.

There are other factors that influence the optimal fee value. For example, due to network delays, the set of unconfirmed transactions may be different from miner to miner. Moreover, when two blocks are discovered at the same time, call them A and B, some users may receive A and then B, others may recieve them in the opposite order, generating a fork. In our simulations, we do not consider these events, but they can be straightforwardly included with the definition of new random variables.

We now introduce a model to estimate the amount of fees collected by a miner over time. Then, we will extend this model to compute whether it is

profitable or not for an attacker to generate a fork when the gap between the value of collected transaction fees and the value of the coinbase transaction reduces.

In addition to the previously introduced simplifications, we also make the following assumptions: 1) the total mining power in the network is constant, and the attackers controls a fraction β of it (while the remaining peers control a fraction $1 - \beta$), and 2) all the miners except for the attacker mine on the main chain (are honest).

12.3.1 Analyzing Transaction Fees

The goal of this first experiment is to see how transaction fees affect the profit of a miner. As already discussed, the total reward for a block is the sum of the fees of the included transactions and the block reward. At the time of the experiments (October 2019), the block reward was 12.5 bitcoin, a value that halves every 210,000 blocks. Currently, as of October 2021, the reward is 6.25 bitcoin. As the number of mined blocks increases, the contribution of the block reward decreases, so fees get a larger influence in the overall profit.

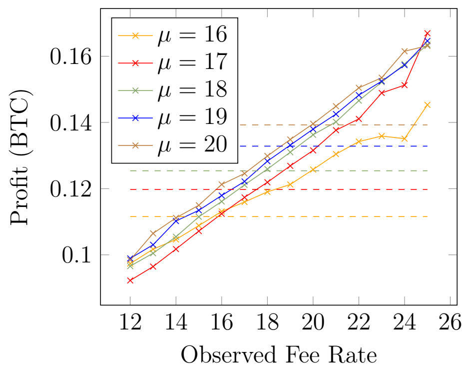
In our model, the block size B and the transactions fee rate R are modelled with normal distributions. The total amount of collected fees is computed as $B \cdot R$. We suppose an average block size of 700kb with variance 50. The value of the average transaction fee rate is retrieved from `blockchain.com` (October 2019), and we suppose a variance of 4. The model is the following:

```
1 block_size(I,BlockSize):gaussian(BlockSize
     ,700,50).
2 fee_rate(I,FeeRate):gaussian(FeeRate,18,4).
3
4 collected_fees(I,O):- block_size(I,B), fee_rate
     (I,R), O is B*R/100000.
```

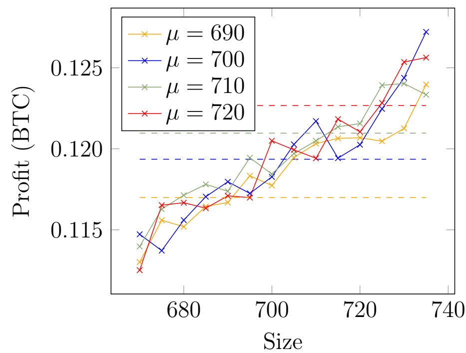
The code works as follows: the predicates `block_size/2` and `fee_rate/2` compute respectively the size of a block and the fee rate associated with a transaction. These values are used in the predicate `collected_fees/2` to get the amount of fees received for the creation of a block (the value is divided by 10^5 since the average fee rate is in satoshi/byte and the block size in kilobyte

but we want the value in bitcoin).

Figure 12.7a shows the expected profit of a miner as a function of the average of the reward, and the expected profit of a miner given that we observed different values of `FeeRate` in `fee_rate/2`. μ represents the mean of the Gaussian distribution modelling the fee rate (in the previous listing, it was fixed to 18). For the fee rate, the variance is fixed to 4 and the observed value `FeeRate` varies between 16 and 20. Values are computed with the predicates `mc_expectation/4` and `mc_lw_expectation/5` from the MCINTYRE package [134]: both sample a certain number of times a variable from the query and return its expectation, but the second also accounts for the evidence and each sample is weighted by its likelihood (likelihood weighting). For both, we set the number of samples to 1000. Figure 12.7b shows how the expected profit of a miner varies by changing the mean of the block size (in the code is fixed to `700`) and by observing different values of `BlockSize` in `block_size/2`. μ represents the mean of the Gaussian distribution modelling the block size (in the previous listing, it was fixed to 700). The variance is fixed to 50. For the reward, we set the mean to 17 and the variance to 4.



(a) Relation between the observed fee rate and the profit of a miner. μ represents the mean of the distribution modelling the fee rate.



(b) Relation between the size of a block and the profit. μ represents the mean of the distribution modelling the block size.

Figure 12.7: Graphs relating the size of the block and the observed fee rate with the profit of a miner. Dashed lines represent the values computed with `mc_expectation/3` (without observations).

In a second experiment, we wanted to compute the probability that a transaction with a certain associated fee is confirmed within a given number of

blocks. This information is crucial for a user since he/she typically wants to attach the optimal fee to it. There are several involved variables: the average number of transactions in a block, the average block discovery time, and the average number of transactions added to the mempool per second. We modelled them as previously discussed. Furthermore, as before, we also considered the average fee rate: this value is modelled with a normal distribution whose mean is re-sampled at each iteration to account for a quick change over time. The obtained model is the following:

```
1  average_fee(_,M):uniform(M,15,25).
2  compute_fee(_,M,F):gaussian(F,M,4).
3  fee(I,F):- average_fee(I,M), compute_fee(I,M,F)
       .
4  compute_time(F,M,V):gaussian(F,M,V).
5  number_of_txs_in_block(_,NB):gaussian(NB
       ,1600,1600).
6  block_discovery_time(_,Time):gaussian(Time
       ,500,500).
7  txs_per_second(_,Txs):poisson(Txs,5).
8
9  generate_pool(N,N,[]):-!.
10 generate_pool(I,N,[F|T]):- I < N, fee(I,F), I1
       is I+1,
11       generate_pool(I1,N,T).
12 get_len(A,B,B):- A >= B, !.
13 get_len(A,B,A1):- A < B, A1 is A-1.
14
15 loop_pool(FeeRate,I,NBlocks,Pool):- I =<
       NBlocks,!,
16       number_of_txs_in_block(I,NB), N11 is round(
           NB),
17       length(Pool,LP), get_len(LP,N11,N1),
18       length(L,N1), append(L,RemPool,Pool),
19       loop_pool_check(FeeRate,I,RemPool,NBlocks).
20 loop_pool_check(_,_,[],_):- !.
```

```
21 loop_pool_check(FeeRate,_,[H|_],_):- H <
      FeeRate,!.
22 loop_pool_check(FeeRate,I,RemPool,NBlocks):- !,
       I1 is I+1,
23     block_discovery_time(I,Time),
24     txs_per_second(I,Txs),
25     NNewTxs is Txs*Time, NT1 is round(NNewTxs),
26     generate_pool(0,NT1,NewArrived),
27     append(NewArrived,RemPool,NewPool),
28     sort(0, @>=, NewPool, PoolSorted),
29     loop_pool(FeeRate,I1,NBlocks,PoolSorted).
30
31 included(I,FeeRate,NBlocks):-
32     loop_pool_check(FeeRate,I,[FeeRate],NBlocks
          ).
```

We first create an initial pool of `N` transactions by sampling using the predicate `generate_pool/3`. Then, the predicates `loop_pool/4` and `loop_pool_check/4` sort the pool in descending order of fees, compute the average number `NB` of transactions in a block, the average block discovery time `Time`, and the average number of transactions per second `Txs`. These values are used to evaluate how many blocks we need to wait before seeing a transaction with fee rate `F` confirmed. The number of new transactions arrived during the last block creation is given by `Txs * Time` (value stored in `NNewTxs`). To simulate the inclusion of transactions in a block, we removed the best (i.e., with highest associated fee rate) `NB` transactions from the pool. If the remaining transaction with the highest fee rate has a value less than `FeeRate` (the target value we consider), this means that our transaction has been successfully included into a block, and the iteration stops. Otherwise, we simulate the arrival of new transactions and repeat the process.

We used the predicates `mc_sample/3` and `mc_lw_sample/4`, from the MCINTYRE package [134] to compute the results. The first samples the goal a certain number of times and returns its probability as the fraction of successes over the total number of samples, while the second, in addition, weights each sample by the evidence. For example, `mc_sample(included(1,17,I),5,P1)`

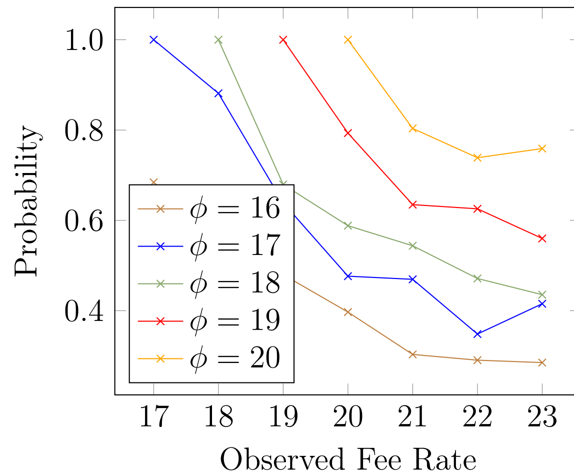


Figure 12.8: The graph shows how transaction fees influence the probability of confirmation within 1 block.

samples 5 times `included(1,17,I)` and returns its probability in `P1`. Similarly, `mc_lw_sample(included(1,17,I),fee(0,18),5,P2)` samples 5 times the predicate `included(1,17,I)` given that `fee(0,18)` has been observed and returns its probability in `P2`. Results are computed with the parameters in the previous code listing are shown in Figure 12.8. `NBlocks` is set to 1, ϕ represents the fees associated with a transaction, and the number of samples is 250. For example, if $\phi = 16$, the query is:

```
mc_lw_sample(included(1,16,1),included(0,ObservedFees,1),250,P)
```

with `ObservedFees` ranging between 17 to 23, as in Figure 12.8. As expected, the probability of a fast confirmation decreases as the value of the observed fees increases. If the value of the associated fee is greater than the observed fee, the probability of inclusion is 1, since the considered transaction will be the most profitable one, and will be included in the next block: for this reason, these values are not reported in the graph.

12.3.2 Probability of a Profitable Fork

Currently, the gap between transaction fees and block reward is still significant. In the next few years, the block reward will decrease, due to the specifications of the protocol. If the number of users increases, the number of transactions will increase as well. Consequently, there will be more competition to quickly

confirm transactions, and fees will probably increase. This will drive to a scenario where the fees and the block reward have similar values. The goal of this experiment is to see whether creating a fork in this situation is profitable.

In the analysis of a double spending attack (see Section 12.2), the model can be split into two parts: the first, where the attacker starts to mine his/her private chain, and the second, where he/she tries to catch up from (possibly) several blocks behind [142]. Here, we focus on the second part, where the attacker tries to catch up from z blocks behind the head. In our experiments, we set z to 1. We can represent this scenario with a one-dimensional random walk, where a particle starts at a given position > 0 and, at each time step, can move one block left or right (see also Section 7.2). Supposing that the attacker controls a fraction β of the total mining power (and all the remaining miners are honest and work all on the same chain), z will increase by one with probability $1 - \beta$ (the honest miners found a block) and decrease by one with probability β (the miner found a block in his private chain). In formulas:

$$z_{t+1} = \begin{cases} z_t + 1 & \text{with probability } 1 - \beta \\ z_t - 1 & \text{with probability } \beta \end{cases}$$

The goal of the attacker is to generate a chain longer than the honest one. If he/she can accomplish this, he/she will publish it, and this would be accepted as the main chain, since it is the longest one. If so, all the transactions not included in this new chain are invalidated. If the attacker controls more than 50% of the total hashing power, he/she will always succeed in the task. In our models, since we are not sure whether the random walk terminates, we suppose that if the gap z between the two chains is greater than 100 the attack fails. In our experiments, we increase the base value of the average fee by a factor of 10. Here, we extend the model presented in [22] and discussed in Section 12.2 with an additional part considering transaction fees, and also discussing some economic implications. The model is the following:

```
1 coinbase(1).
2 move(T,P1,1):1-Beta; move(T,P1,-1):Beta:- Beta
     is P1/100.
3
```

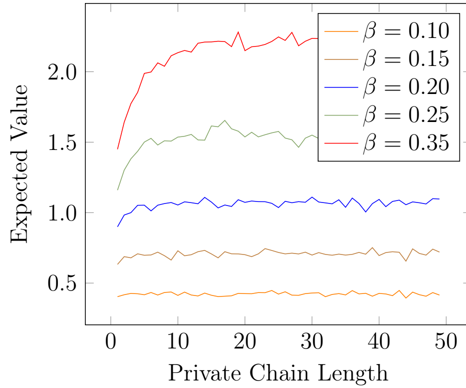
```
 4 walk(Z,S,BetaPercentage,ThresholdMinedBlocks,
      TotalMinedByAttacker):-
 5      walk(Z,0,S,BetaPercentage,
           ThresholdMinedBlocks,0,
           TotalMinedByAttacker).
 6
 7 walk(-1,S,S,_,_,V,V).
 8 walk(Z,T0,S,BetaPercentage,ThresholdMinedBlocks
      ,MinedBlocks,VT):-
 9      Z >= 0,
10      MinedBlocks < ThresholdMinedBlocks,
11      Z < 100,
12      move(T0,BetaPercentage,Move),
13      T1 is T0+1,
14      Z1 is Z+Move,
15      ( Move < 0 ->
16          V1 is MinedBlocks + 1;
17          V1 = MinedBlocks
18      ),
19      walk(Z1,T1,S,BetaPercentage,
           ThresholdMinedBlocks,V1,VT).
20
21 reward_fork(BetaPercentage,ThresholdMinedBlocks
      ,ExtraValue,ExpectedReward):-
22      coinbase(C),
23      fee(1,AvgFee),
24      ExtraVal is AvgFee*ExtraValue,
25      (   walk(1,_,BetaPercentage,
           ThresholdMinedBlocks,Mined) ->
26          fee(2,Fees),
27          ExpectedReward is (C+Fees)*Mined +
              ExtraVal;
28          ExpectedReward is 0
29      ).
```

The predicate `reward_fork/4` is used to compute the reward. It calls `walks/5` to simulate a one-dimensional random walk. `walk/5` wraps `walk/7` that imposes the maximum gap of 100 blocks between the two chains. The predicate `move/3` increases or decreases this gap. The run continues until the attacker succeeds or the gap is too big. `fee/2` is the same predicate discussed before (Section 12.3.1).

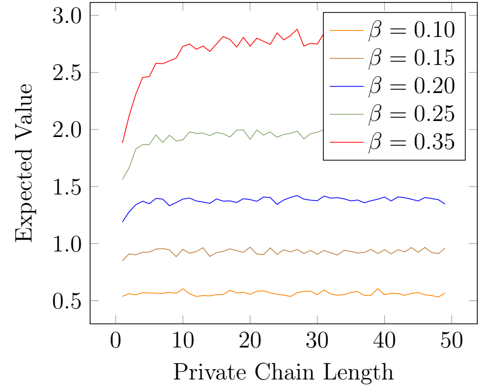
In the first experiment (threshold experiment), the goal is to see how the controlled hashing power and the length of the private chain influence the possible reward. For all the experiments, the average total reward for a block (coinbase + fees) is normalized to 1, and the extra value stored in an already mined block on the honest chain (variable `ExtraValue` in the previous listing) is indicated with σ. If $\sigma = 0.5$, this means that the block from which the fork starts has a 50% larger reward than the average.

Figure 12.9 shows the results of our experiments computed with the predicate `mc_expectation/4` with 10000 samples (also the values for the next two experiments will be computed with the same predicate and the same number of samples). The graphs in figures 12.9a and 12.9b are obtained by considering a successful attack where the attacker was able to create a chain with length equal to the main chain. Figures 12.9c and 12.9d consider a successful attack where the private chain is 1 block longer than the main chain. Graphs in figures 12.9a and 12.9c have $\sigma = 1$ while graphs in figures 12.9b and 12.9d have $\sigma = 2$. In particular, there is a value for each β where the expected value settles. Clearly, if a miner controls a high fraction of the total hashing power, the probability of success will be high. The larger is his/her power, longer his/her private chain should be to get the maximum available reward. The optimal length of the private chain does not change as the extra value in a block increases.

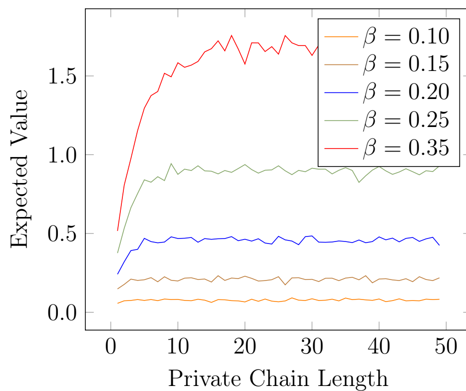
In a second experiment (value experiment), we modelled the possible profit gained by re-mining a block with extra value σ. The computed values are reported in Figure 12.10. The graphs in figures 12.10a and 12.10c have $\beta = 0.15$ while graphs in figures 12.10b and 12.10d have $\beta = 0.25$. Figures 12.10a and 12.10b contain values computed considering a successful attack where the number of blocks for the two chains is the same, while in figures 12.10c and 12.10d we have a scenario where the attacker succeeds only if its private



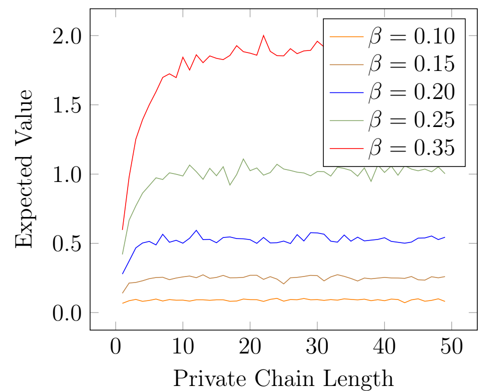
(a) Expected gained profit of an attacker that is able to create a chain with length equal to the main chain starting from a block with twice average reward ($\sigma = 1$).



(b) Expected gained profit of an attacker that is able to create a chain with length equal to the main chain starting from a block with three times the average reward ($\sigma = 2$).



(c) Expected gained profit of an attacker that is able to create a chain with one more block than the honest chain starting from a block with twice average reward ($\sigma = 1$).

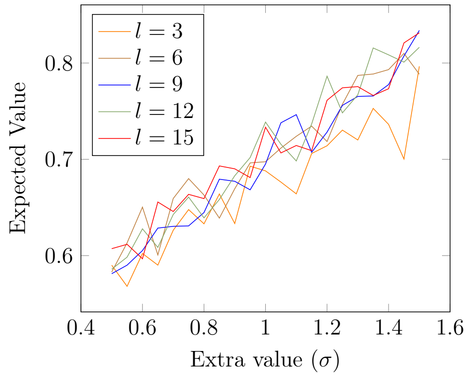


(d) Expected gained profit of an attacker that is able to create a chain with one more block than the honest chain starting from a block with three times the average reward ($\sigma = 2$).

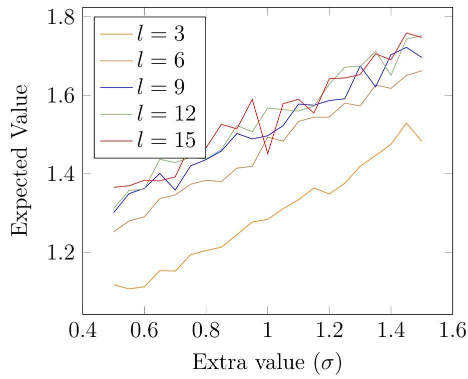
Figure 12.9: Results for the threshold experiment.

chain is one block longer than the honest one. Results show that if an attacker gives up too early, his/her expected reward can be lower than the maximum.

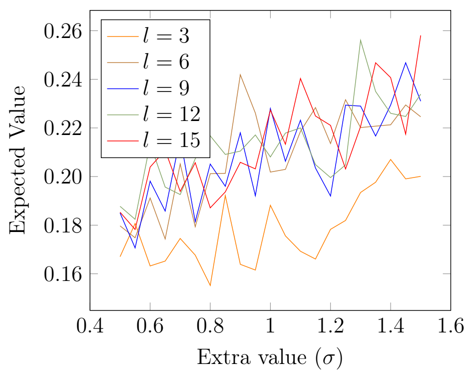
Finally, we computed the difference between the expected value obtained from being honest and from forking the chain (optimal experiment). The expected mining reward is the expected value of a binomial distribution with probability of success β, where β is the fraction of the controlled hashing power.



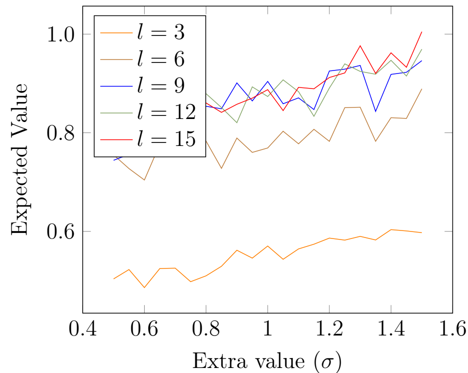
(a) Expected profit gained by re-mining a block with variable extra value σ controlling 15% of the total hashing power ($\beta = 0.15$). The attack is successful if the length of the private chain is equal to the honest chain.



(b) Expected profit gained by re-mining a block with variable extra value σ controlling 25% of the total hashing power ($\beta = 0.25$). The attack is successful if the length of the private chain is equal to the honest chain.



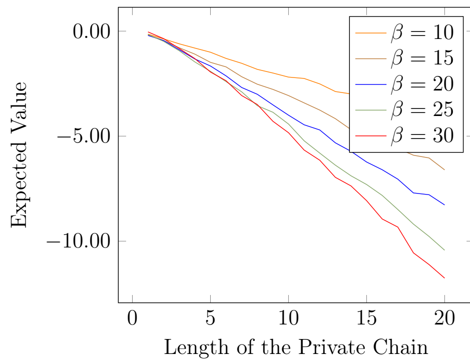
(c) Expected profit gained by re-mining a block with variable extra value σ controlling 15% of the total hashing power ($\beta = 0.15$). The attack is successful if the private chain is one block longer than honest chain.



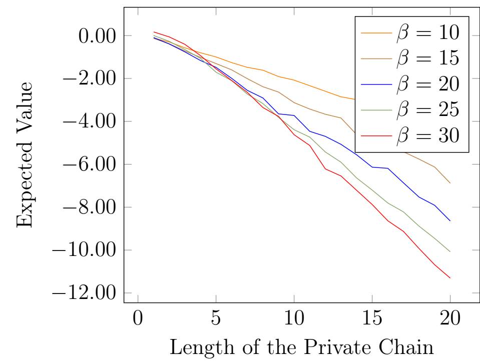
(d) Expected profit gained by re-mining a block with variable extra value σ controlling 25% of the total hashing power ($\beta = 0.25$). The attack is successful if the private chain is one block longer than honest chain.

Figure 12.10: Results for the value experiment.

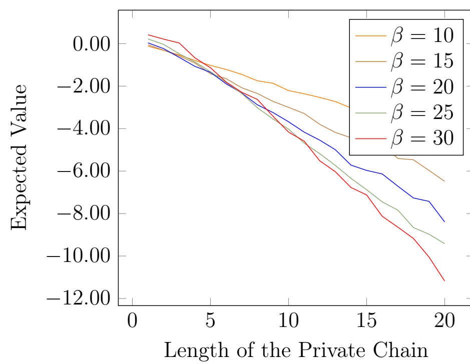
We are interested in whether there exists an extra value σ that makes a fork profitable and whether keeping mining a private chain is still feasible when the gap with the honest chain is substantial. As Figure 12.11 shows, this is feasible only for significant values of σ and β.



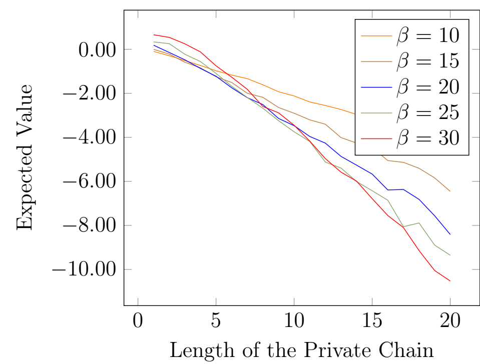
(a) $\sigma = 2$.



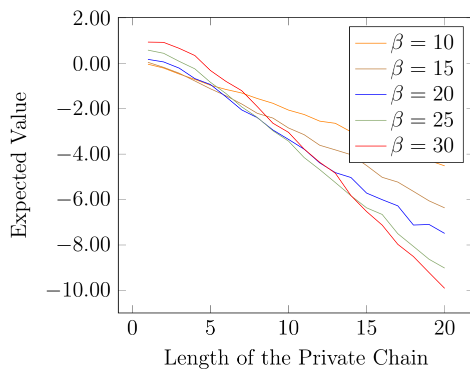
(b) $\sigma = 4$.



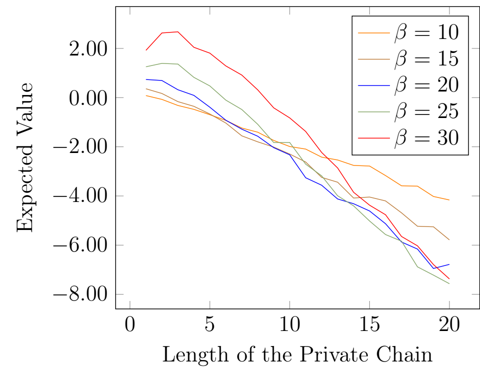
(c) $\sigma = 6$.



(d) $\sigma = 8$.



(e) $\sigma = 10$.



(f) $\sigma = 20$.

Figure 12.11: Results for the optimal experiment. Each graph shows the difference between the expected value obtained from being honest and from forking the chain starting from a block with n times the average reward (σ) with different values of the fraction of the controlled mining power β.

12.3.3 Related Work

There are several related articles on the study of Bitcoin transaction fees. In [92, 98], the authors adopt queuing theory to study the relation between fees and confirmation time. In particular, in [92] the authors show that a transaction with a small amount of associated fees requires a large confirmation time if there is an increasing number of arrivals of transactions with fees smaller than a certain threshold. Moreover, they state that an increase of the block size will not be effective in reducing the confirmation time. There are some studies that introduce new methods to avoid the fluctuation of the fees, such as [30]. The authors of [161] proposed a game theory model to investigate fees and observed that the current state of the system (as of 2018) incentivizes the formation of large mining pools. An analysis on how block reward, transaction fees and their ratio influence the Bitcoin ecosystem can be found in [42]. In [104], the authors analyse a "whale attack", where a user issues transactions with large fees. In [113] the authors provide an in-depth analysis of transaction fees looking at historical data. In [22, 124, 142] the authors analyse the double spending attack and show that it is profitable only for high fractions of controlled mining power. Finally, there are some works that analyse the economic influence of bitcoin and cryptocurrencies [54, 94, 172] also by looking at Google Trends and Wikipedia data [67, 101].

12.3.4 Conclusions

In this section, we proposed several PLP models to study transaction fees. In particular, we analysed how fees influence the revenue of a miner, also in the case when the difference between block reward and collected fees reduces.

Our result shows that, as expected, an increase in the block size will increase as well the revenue also when the gap between fees and block reward reduces. However, this situation requires a more accurate analysis since bigger blocks will cause several problems, such as network delays due to slower propagation times and consequently an increasing amount of (unintentional) forks.

We then extended the previous model to see whether it is profitable for an attacker to fork the main chain to mine again a block with particularly high fees attached. Our experiments state that this is profitable only if both the

hashing power of the attacker and the fees associated with the block are of considerable amount. If the computing power is relatively small, the attack is profitable only if it succeeds within a few blocks.

In general, our models show that the system is reliable even if the gap between block reward and collected fees reduces, and forks are not profitable, even for mining pools. This is a desirable property since it disincentivizes attacks and preserves the consistency and reliability of the system. However, when the fees overcome the block reward by a significant amount, mining pools could be economically incentivized to deviate from the honest chain in some cases (even if it is very unlikely to happen).

Finally, our models can be extended in several ways. For example, they can be integrated with other variables, as discussed at the beginning of this section. Moreover, decision theory models can be applied to compute the optimal set of transactions instead of the probability.

12.4 Lightning Network Model

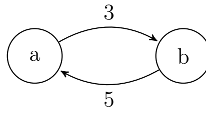
We can see the LN as a graph where users (identified with nodes) are connected through edges with a certain capacity. A connection between node `A` and node `B` with capacity `Capacity` is denoted with the fact `edge(A,B,Capacity)`. The whole network is represented with a set of `edge/3` facts. For simplicity, in this model we do not consider fee base and fee rate. However, they can be easily included by adding two more arguments to `edge/3`. Payments can go in both directions, so we represent them using the predicate `connected/3`

```
1 connected(A,B,C):- edge(A,B,C) ; edge(B,A,C).
```

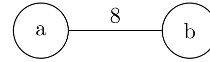
that states that `A` and `B` are connected through a channel of capacity `C` if there is an edge from `A` to `B` or from `B` to `A` with that capacity. This is also depicted in Figure 12.12.

There can be multiple edges with the same source and destination. The degree of a node is the number of edges incident to that node. We can search for a path between two nodes with the following code:

```
1 connected_test(Source,Next,Size):-
2     connected(Source,Next,Cap),
```



(a) Two directed edges.



(b) One undirected edge.

Figure 12.12: Channel representation in the Lightning Network. Case (a) corresponds to (b) in practice since the distribution is unknown.

```
3       Size < Cap.
4
5   path(Dest,Dest,_,_,Path,Path).
6   path(Source,Dest,Size,NSteps,Visited,Path) :-
7       length(Visited,N),
8       N < NSteps,
9       connected_test(Source,Next,Size),
10      \+ memberchk(Next,Visited),
11      path(Next,Dest,Size,NSteps,[Next|Visited],
            Path).
```

The predicate `path(Source,Dest,Size,NSteps,Visited,Path)` is true if there is a path from `Source` to `Dest` with capacity at least `Size`. Nodes and edges cannot be traversed twice. Line 10 checks this requirement with `memberchk/2` that is a deterministic version of `member/2`, a standard Prolog predicate that is true if the first argument is present in the list passed as the second argument. This is fundamental since, otherwise, we may get stuck in a loop due to a perpetual visit of the same pair of nodes. We limit the length `N` of the path with the standard Prolog comparison predicate `</2`. Similarly, we can check whether a connection has enough capacity `Cap` to route a payment of size `Size` with `connected_test/3`. A sample call to `path/6` could be `path(a,d,11,3,[],P)`: we search for a path `P` from `a` to `d` of at most `3` edges that can route a payment of size `11`, starting with an empty list (`[]`) of visited nodes.

12.4.1 Deterministic Model

We conducted several experiments to study the LN. We selected three successive snapshots representing its state on 2nd August, 3rd August, and 4th Au-

gust 2020[8] [141]. Table 12.5 shows some statistics about these three datasets.

Dataset	# Nodes	# Edges	Avg. Edge Capacity	Total Capacity
February	4,562	30,249	2,761,826.794	83,542,498,678
March	4,709	30,607	2,839,773.151	86,916,936,824
April	4,838	30,400	2,935,025.096	89,224,762,929

Table 12.5: Datasets structure for the three LN states (February, March, and April). Capacities are expressed in satoshi.

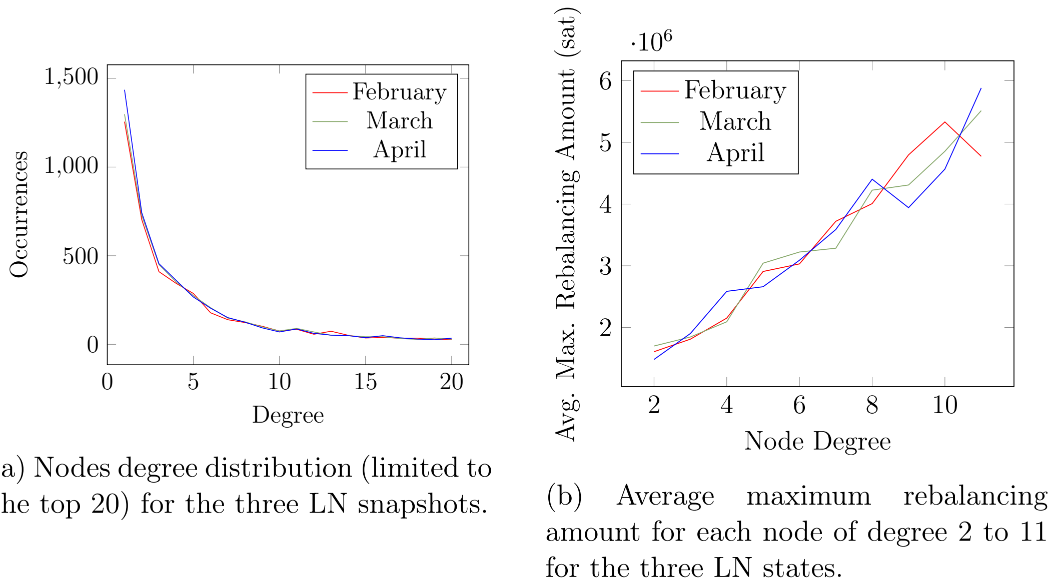
Each node can obtain a global vision of the network through the broadcast messages `channel_announcement` and `channel_update`. The obtained data are used to construct a path for a payment between nodes not directly connected with an edge. However, there can be hidden payment channels, not publicly announced, so there is no guarantee that the information about the possible connections is complete.

We wrote the code using SWI-Prolog [171] and conducted several experiments to describe the temporal evolution of the LN, focusing on several aspects. First, we computed the node degree distribution (Figure 12.13a): most of the nodes of the network have degree between 1 and 5 (more than 65% for all three datasets). Then, we focused on the maximum possible rebalancing (source and destination of the payment coincide) amount (Figure 12.13b): as expected, as the degree of the nodes increases, the maximum rebalancing amount increases as well. The computation took 2.39 hours to 6.92 hours for the February network, 2.53 hours to 7.31 hours for the March network, and 2.65 hours to 7.1 hours for the April network on the machine described in Section 7.2.1.

We gathered the frequency of the most common capacities associated with edges and reported them in Table 12.6. Results for all three instances are almost the same, except for the fourth and the fifth positions in March, where these are swapped with respect to February and April.

We computed the variation of the total capacity of the network (sum of the capacities of all the edges) by removing the edges with the highest 50 capacities and the nodes associated with the highest 100 degrees. The goal is to see how much of the total capacity is in the hands of a few. Figure 12.14a shows that, by removing edges, the slope of the curve substantially reduces

[8]Datasets available at `https://gitlab.tu-berlin.de/rohrer/discharged-pc-data`



(a) Nodes degree distribution (limited to the top 20) for the three LN snapshots.

(b) Average maximum rebalancing amount for each node of degree 2 to 11 for the three LN states.

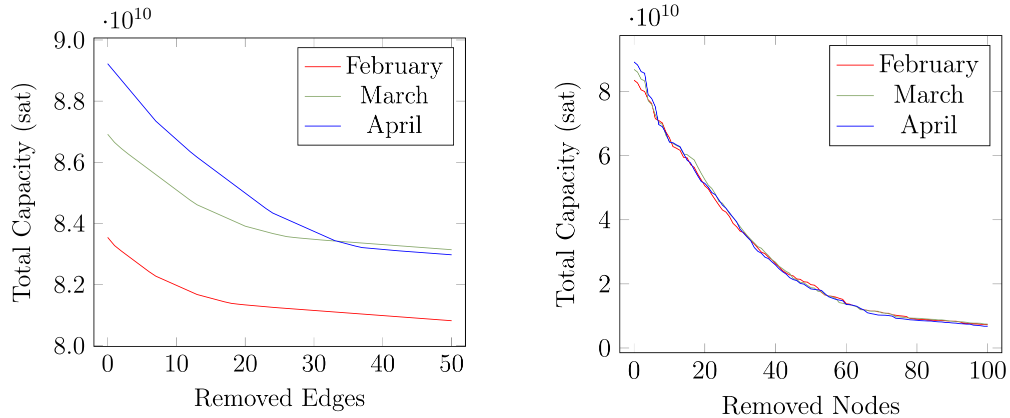
Figure 12.13: Nodes degree distribution and average maximum rebalancing amount.

after 50 removals. Moreover, removing the 100 nodes with the highest degrees (Figure 12.14b) decreases the capacity of approximately 90%.

We considered the number of paths of length 2 and 3 involving nodes with the same degree. We varied the degree between 1 and 10. We focused on short paths since, on average, the length of the shortest path between two nodes is 2.8 [153]. Furthermore, longer paths also imply higher fees to pay, since the payments require additional intermediate steps. This query was performed on GNU/Linux machines with Intel Xeon E5-2697 v4 (Broadwell) at 2.30 GHz,

Frequency (F)	Capacity (F)	Frequency (M)	Capacity (M)	Frequency (A)	Capacity (A)
2,713	1,000,000	2,709	1,000,000	2,595	1,000,000
1,883	100,000	1,865	100,000	1,931	100,000
1,780	500,000	1,749	500,000	1,735	500,000
1,425	2,000,000	1,465	16,777,215	1,460	2,000,000
1,346	16,777,215	1,459	2,000,000	1,444	16,777,215
1,271	5,000,000	1,298	5,000,000	1,199	5,000,000
869	200,000	846	200,000	814	200,000
788	20,000	777	20,000	789	20,000
594	10,000,000	599	10,000,000	648	10,000,000
573	300,000	581	300,000	581	300,000

Table 12.6: Capacity of the edges (expressed in satoshi) and frequency for February (F), March (M), and April (A).



(a) Variation of the total network capacity by removing the top 50 edges with the largest capacity.

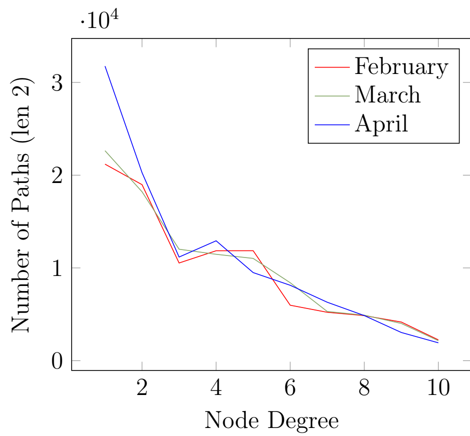
(b) Variation of the total network capacity by removing the top 100 nodes with the highest degree.

Figure 12.14: Variation of the total network capacity by removing edges and nodes for the three LN states.

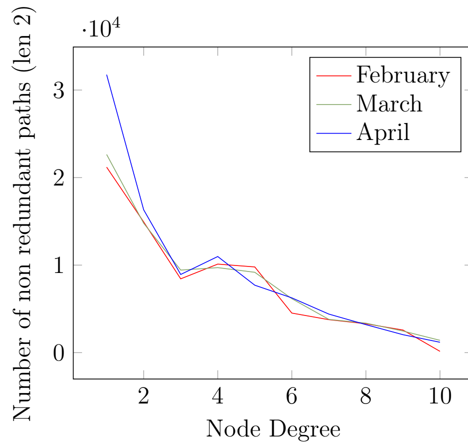
with a maximum allowed time of 24h, as it was very time-consuming. We were not able to compute the number of paths for a length greater than 3 as the query exceeded 24 hours. The computation of the number of paths of length 2 took 11.72 minutes, 11.58 minutes, and 13.56 minutes for the February, March, and April network respectively. The computation of the number of paths of length 3 took 13.42 hours, 15.14 hours, and 15.58 hours on the February, March, and April network respectively. Figures 12.15a and 12.15c show that the number of paths substantially decreases after the 3rd or 4th degree for the three months for both paths of length 2 and 3.

Finally, we also considered the number of non-redundant paths of length 2 and 3 involving nodes with the same degree between 1 and 10. Differently from the previous query, here we do not consider redundant paths, meaning that if two nodes are connected with more than one path, its contribution to the total value is still 1. Figures 12.15b and 12.15d show the results. The computation of the number of paths of length 2 took 9.15 minutes, 10.08 minutes, and 10.75 minutes while for paths of length 3, 4.57 hours, 5.01 hours, and 5.71 hours for February, March, and April network respectively, on the same machine previously described.

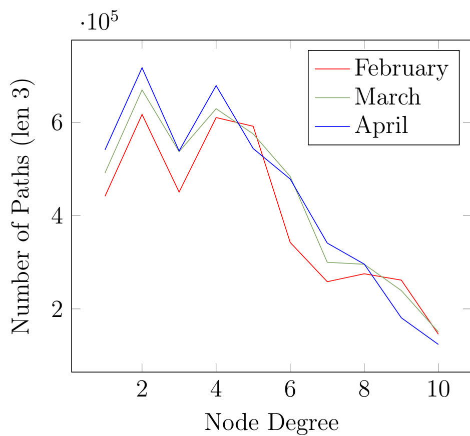
Overall, we proposed an approach to compute different LN properties using Logic Programming. However, there is uncertainty on the routing in LN, since,



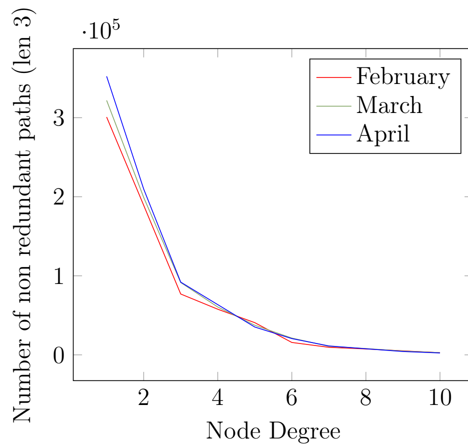
(a) Number of paths of length 2 between equal degree nodes (for variable degree).



(b) Number of non redundant paths of length 2 between equal degree nodes (for variable degree).



(c) Number of paths of length 3 between equal degree nodes (for variable degree).



(d) Number of non redundant paths of length 3 between equal degree nodes (for variable degree).

Figure 12.15: Number of paths and non redundant paths of length 2 and 3 between equal degree nodes.

for example, some nodes may not be active, or refuse to forward a payment. Moreover, the distribution of the funds in a channel is unknown. For all these reasons, in the next section we see how this deterministic model can be extended to a probabilistic one.

12.4.2 Probabilistic Model

As discussed in Section 11.1.3, the balance distribution of a channel is unknown, so the task of routing transactions can be seen as probabilistic. Thus, we can represent the capacity of a channel using a probability distribution. To do this, we can extend the previously discussed code with a probabilistic fact to model the capacity of a channel with a uniform distribution between `L` and `U` (variables that will be bound to numbers). We can do this using the `cplint` hybrid program syntax (see Section 8.2.1) with:

```
1 distr(X,L,U) : uniform_dens(X,L,U).
```

Consequently, the predicate `connected_test` can be modified as:

```
1 connected_test(Source,Next,Size):-
2     connected(Source,Next,Cap),
3     distr(C,0,Cap),
4     Size < C.
```

With this modification, we first collect the total capacity `Cap` of the channel between two nodes, and then we state that the amount `C` between `Source` and `Next` is uniformly distributed between `0` and `Cap`. The predicate succeeds if the obtained value is larger than the payment size. Moreover, at each iteration, the distribution of funds for the same edge does not change, i.e., if we sample a path between two nodes, the distribution of funds in all the considered channels remains the same.

The model can be further extended by considering intermittent edges. This can be due to nodes that disconnect from the network or refuse to forward a payment. To model this, we define a Bernoulli random variable that is true with a certain probability, for example 0.95, with:

```
1 active(_):0.95.
```

With this addition, the predicate `connected_test/3` becomes:

```
1 connected_test(Source,Next,Size):-
2     connected(Source,Next,Cap),
3     active(Next),
4     distr(C,0,Cap),
5     Size < C.
```

We choose a Bernoulli distribution for intermittent edges and a uniform distribution for the capacity of the edges because we do not have further information. However, several extensions are possible: using a Gaussian distribution with the mean equal to half of the capacity of the channel, or the probability that a node refuses a payment proportional to its size, associated fees, or a combination of the two.

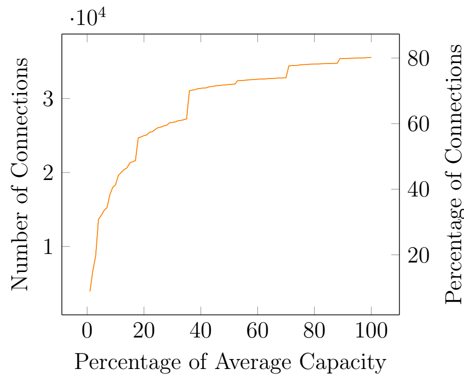
To conduct our experiments, we used a LN snapshot taken on 12th April 2021[9]. We selected only the open channels, resulting in a graph with 14734 nodes, 44349 edges, and a total capacity of 125819660675 satoshi (1258.19 bitcoin). Table 12.7 shows a summary of the most common channel capacities and node degrees.

Most Common Capacities	Occurrences	Degrees	Occurrences	Highest Capacities	Occurrences
100000	3942	1	6958	500000000	4
1000000	3523	2	2575	477184791	1
500000	2794	3	1372	354000000	1
2000000	1533	4	802	300000000	2
16777215	1522	5	549	250000000	1
200000	1446	6	397	238135604	1
5000000	1359	7	263	225118006	1
20000	1297	8	209	200000000	28
10000000	1131	9	182	179792707	1
50000	1079	10	145	154222260	1

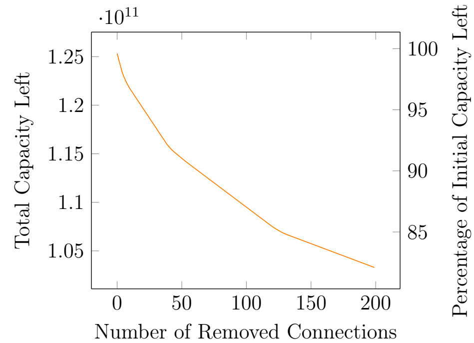
Table 12.7: Information about channel capacities (in satoshi) and node degrees.

We conducted some experiments to prove the effectiveness of PLP in modelling the LN. To choose the range of the payment size, we first computed the average of the capacities of all the connections, obtaining approximately 2837035 satoshi but with a standard deviation greater than 10^7. So, we counted the number of connections that have less than the average capacity, less than half of the average capacity, and less than a quarter of the average capacity, obtaining respectively 35555 ($\approx$ 80%), 31902 ($\approx$ 72%), and 26077 ($\approx$ 59%). Results are shown in Figure 12.16a, where the X axis indicates the percentage of the average capacity and the Y axes indicate the number of connections (edges) with less than that value and the relative percentage with respect to the total connections. To further analyse the distribution of the capacity, we removed the nodes with the highest capacities and plotted the variation of the total capacity. The results are shown in Fig 12.16b.

[9]Snapshot taken from `https://ln.bigsun.xyz/`.



(a) Number and percentage of connections with less than a certain percentage of the average capacity (2837035).

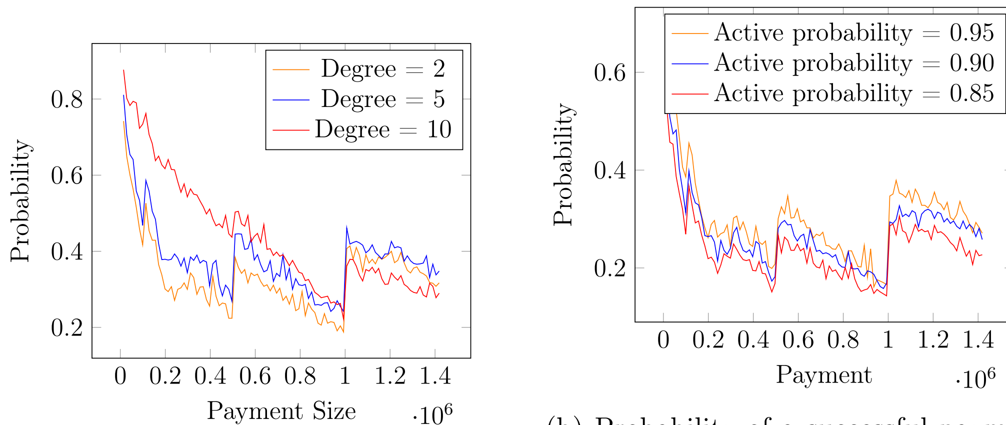


(b) Value and percentage of capacity left after removing the top n (X axis) connections in terms of capacity.

As a first test, we computed the probability to successfully route a payment of varying size at the first attempt between two random different nodes of the same degree. Some nodes along the path may not be active. Clearly, a payment succeeds if all the intermediate nodes are active and forward the payment. We fixed the number of intermediate edges to 2 and fixed the degrees of the source node and the destination node to 2, 5, and 10 for three different tests. The goal is to see how the probability varies when intermittent nodes may not be active. Probabilities are computed using the predicate `mc_sample(+Query:atom,+N:int,-Prob:float)` from the MCINTYRE module [134] with 1000 samples.

As Figure 12.17a shows, nodes with higher degrees have, as expected, a higher probability to successfully route a payment at the first attempt. However, this gap reduces as the payment size increases. When nodes may be disconnected, as shown in Figure 12.17b, the probability of a successful routing decreases, but not so drastically.

We conducted another experiment where we computed the probability of a successful payment split into various equal parts between two random nodes of variable degree. The payment size is split into N parts, and we computed the probability that all these N parts are successfully routed at the first attempt. This is a common scenario [123] since edges have limited capacity and a payment of considerable size has low probability of being accepted by intermediate nodes. However, splitting the payment into several parts increases the fees needed to route it, since each sub-payment must be sent independently



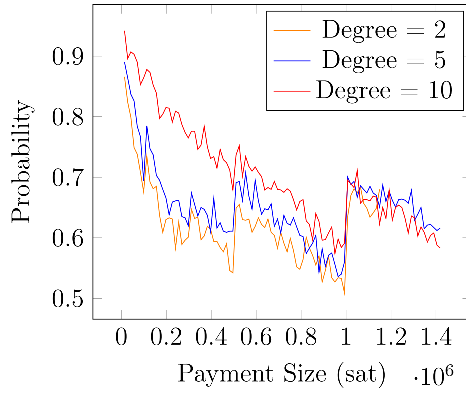
(a) Probability of a successful payment of varying size between random connected nodes of degree 2, 5, and 10.

(b) Probability of a successful payment of varying size between random connected nodes of degree 2, with different active probabilities for intermediate nodes.

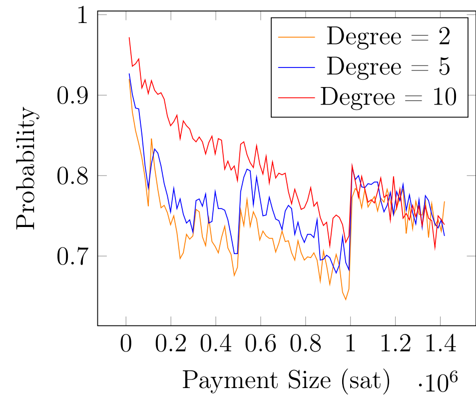
Figure 12.17: Probability of a successful payment of varying size between random nodes.

and has an associated fee. The graphs in figures 12.18a, 12.18b, and 12.19a show how the probability of a successful payment varies when it is split in 2, 3, or 4 parts and intermediate nodes are always active. In figures 12.19b, 12.20a, and 12.20b we repeated the experiment with a fixed degree of the source and the destination (2), but we varied both the probability that a node is active and the number of parts of a payment.

All the plots have a clear jump around 500000 ($5 \cdot 10^5$) and 1000000 (10^6) satoshi. This is probably due to the distributions of the capacities of the edges. In fact, these two values are the second and third most common capacities, as shown in Table 12.7. In our experiments, source and destination are randomly chosen and we select a path if all the intermediate edges have enough total capacity to route a payment of a specified size. If so, we proceed with the probabilistic analysis, otherwise we search another path. However, if the value of the payment is close to the total capacity of one of the edges of the path, the routing will likely fail since we assumed a uniform distribution of the capacity. For example, the routing of a payment of size $4.5 \cdot 10^5$ through a channel of capacity $5 \cdot 10^5$ has only approximately 10% of chances to succeed (with a uniform distribution). Many channels have capacity $5 \cdot 10^5$ so, when the payment size gets closer to this value, the probability of a successful routing at

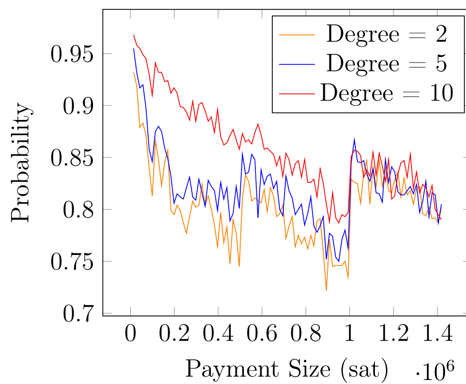


(a) Probability of a successful payment of varying size split into 2 equal parts, between connected nodes of various degrees.

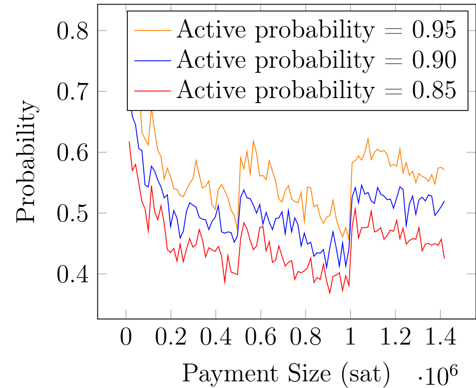


(b) Probability of a successful payment of varying size split into 3 equal parts, between connected nodes of various degrees.

Figure 12.18: Probability of a successful payment split in multiple parts between nodes of various degrees where intermediate nodes are always active.



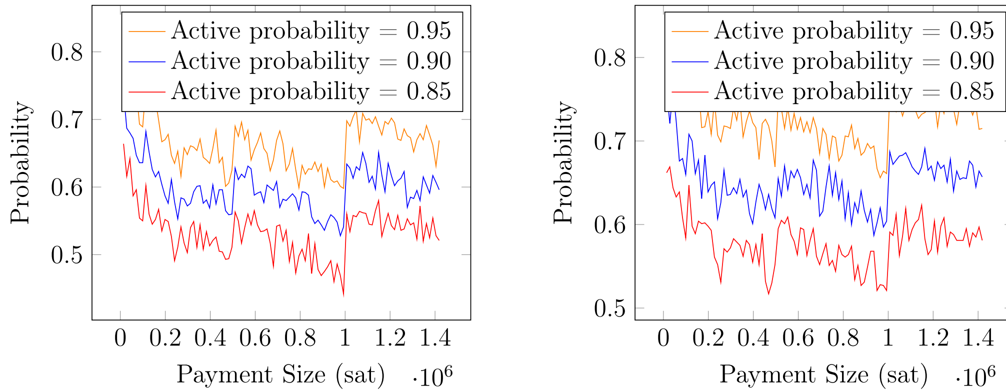
(a) Probability of a successful payment of varying size split into 4 equal parts, between connected nodes of various degrees.



(b) Probability of a successful payment of varying size split into 2 equal parts, between connected nodes of degree 2 with varying active probabilities.

Figure 12.19: Probability of a successful payment split in multiple parts between nodes of various degrees.

the first attempt decreases. If the size is greater than $5 \cdot 10^5$, these channels are not considered since they do not have enough funds. Similarly for the value 10^6. There is another but less noticeable jump around 100000 (10^5), which is the most common value for the capacity of a channel. This happens for the same reasons just explained.



(a) Probability of a successful payment of varying size split into 3 equal parts, between connected nodes of degree 2 with varying active probabilities.

(b) Probability of a successful payment of varying size split into 4 equal parts, between connected nodes of degree 2 with varying active probabilities.

Figure 12.20: Probability of a successful payment between nodes of various degrees when intermediate nodes could be inactive.

The goal of these experiments was to prove the feasibility modelling the LN with PLP, rather than provide an in-depth analysis of the LN, since it is very dynamic and its structure rapidly changes, making all the computed values quickly obsolete. Moreover, routing mechanisms are more sophisticated than random routing, which we adopted here. However, our experiments show that a PLP model is a useful representation of the network and can help to study in depth several possible scenarios.

12.5 Conclusions

In this chapter, we constructed several probabilistic logic models that encode different possible situations that may happen in a blockchain environment. In Section 12.1, we provided a probabilistic logic analysis of Solidity smart contracts and leveraged this to compute different profit values. In Section 12.2, we presented two models regarding the centralization of the hashing power and the double spending attack, proving their effectiveness. In Section 12.3, we wrote a PLP model to represent how transaction fees can influence the whole blockchain ecosystem, and studied whether for some miners it is profitable to be dishonest. Finally, in Section 12.4, we provided two models for the LN, one

deterministic and one probabilistic, and computed several probability values regarding the routing of the payments. Overall, PLP is a powerful formalism for many application scenarios thanks to its expressivity combined with the simplicity of the language.

Part V

Conclusions and Outlooks

Chapter 13

Conclusions

The work presented in this dissertation focuses on the extensions and applications of Probabilistic Logic Programming (PLP). In the first two parts we introduce respectively the needed background knowledge regarding mathematical and logical concepts and already existing PLP languages and techniques to perform inference. In Part II, we also provide a novel implementation of a Markov Chain Monte Carlo algorithm to perform approximate inference in PLP. Part III discusses our newly proposed extensions for PLP:

- Hybrid probabilistic logic programs, where discrete and continuous random variables coexist. We start from an overview of the existing languages, with a particular focus on Probabilistic Constraint Logic Programming. Then, we provide a new formal semantics and prove it well-defined. Finally, we discuss several syntactic requirements to preserve its properties.

- Probabilistic abductive logic programs, where PLP is extended with abduction to manage uncertain and incomplete data. We provide a formal definition of this new class and a practical algorithm to perform inference.

- Probabilistic optimizable logic programs, where some facts can be marked as "optimizable" and their probabilities can be set. The goal is to set the optimal values of these facts such that an objective function is optimized and the constraints on probabilities of optimizable facts are satisfied, a task that can be considered as parameter learning under constraints. To perform reasoning, we propose an algorithm that extracts an equation

from a program and then leverages a constraint solver to compute the solution.

- Probabilistic reducible logic programs, where, similarly to probabilistic optimizable logic programs, some probabilistic facts can be marked as "reducible" and they can be removed from the program. Here, the goal is to remove as many reducible facts as possible while keeping the validity of some constraints involving random variables values. This task can be considered as structure learning under constraints. As before, we propose a reasoning algorithm that works by extracting an equation and passing it to a constraint solver. These last two extensions represent a step towards the integration of constraints and probabilities.

The introduction of these new extensions is motivated by several examples discussed in the correspondent sections.

In part IV, we apply the PLP formalism to the blockchain ecosystem. After introducing basic blockchain concepts, especially focusing on Bitcoin, we analyse:

- Smart contracts, i.e., programs executed on a blockchain: we provide a possible PLP encoding of them and discuss its benefits.

- Hashing power centralization and double spending, two possible situations that can happen in a blockchain environment: we implement two already existing models and see how PLP can be useful to test their robustness.

- Transaction fees and how possible future scenarios can generate competition between users and miners.

- Lightning Network, a layer-two solution with the goal of increasing the possible transactions per second and limiting some existing problems in Bitcoin: starting from real snapshots of the network, we compute several probability values involving payment routing. We consider both a deterministic and a probabilistic model where users can disconnect from the network.

Chapter 14

Future Work

The PLP field is vastly under-explored, so there are several possible future streams of work involving different topics.

In this dissertation, we used an already proposed encoding for PLP that leverages binary decision diagrams. A future work can be the exploration of several alternative languages for knowledge compilation and see how they relate in terms of compactness and expressivity. Moreover, newly proposed works suggest a matrix encoding for a logic program: it can be interesting to extend this also to probabilistic logic programs with the goal of leveraging existing deep learning libraries operating on matrices.

Lifted inference is a technique for performing inference at a lifted level to avoid grounding as much as possible. However, this technique can be applied only to some classes of programs that present a particular structure. There are few works that applied this concept to PLP and more work is needed.

Approximate inference is usually unavoidable when managing real-world domains. In this dissertation we introduced two Markov Chain Monte Carlo algorithms that mimics Metropolis Hastings and Gibbs sampling. However, in the probabilistic programming literature there is a plethora of alternative solutions. An interesting direction would be to provide an implementation of these also for PLP. Moreover, a future work could consist in developing algorithms for approximate inference that can be applied to our newly introduced classes of programs.

We introduced a semantics for hybrid probabilistic logic programs and discussed the syntactic requirements, but a practical inference algorithm is miss-

ing. This would be of much interest given the expressivity of the language, but its implementation is not trivial since the domain is infinite.

There are several possible directions in terms of modelling blockchain scenarios with PLP. For example, as discussed in Section 12.1, a smart contract language based on logic will be of much interest, since it will permit using directly the PLP model we proposed. In the context of the Lightning Network, it would be interesting to extend the model discussed in Section 12.4 using, for example, inductive logic programming techniques to guess where possible hidden nodes can be or where new nodes can be positioned. Moreover, an integration with the frameworks described in sections 10.1 and 10.2 would be interesting. This will allow to remove some of the possible connections, or tune their funds, while keeping the probability of successfully routing a payment between some lower and upper bounds.

Neural-symbolic integration is currently one of the (if not the) hottest topics in machine learning. Combining PLP with deep learning (neural networks) will allow to integrate the expressivity of the former with the scalability of the latter. Despite the enormous amount of research in this area, the proposed solutions are still limited, and there is still disagreement on how these two paradigms should be integrated. PLP could be one of the core components of a possible integration, so work in this direction can be very exciting.

Bibliography

[1] Marco Alberti, Elena Bellodi, Giuseppe Cota, Evelina Lamma, Fabrizio Riguzzi, and Riccardo Zese. Probabilistic constraint logic theories. In Arjen Hommersom and Samer Abdallah, editors, *Proceedings of the 3rd International Workshop on Probabilistic Logic Programming (PLP)*, volume 1661 of *CEUR-WS*, pages 15–28, Aachen, Germany, 2016. Sun SITE Central Europe.

[2] Marco Alberti, Elena Bellodi, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese. `cplint` on SWISH: Probabilistic logical inference with a web browser. *Intelligenza Artificiale*, 11(1):47–64, 2017.

[3] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the $\mathcal{S}$CIFF framework. *ACM Transactions on Computational Logic*, 9(4):29:1–29:43, 2008.

[4] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, Inc., 2014.

[5] Valentin Antuori and Florian Richoux. Constrained optimization under uncertainty for decision-making problems: Application to real-time strategy games. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 458–465, 2019.

[6] Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3-4):335–363, 1991.

[7] Krzysztof. R. Apt, Howard. A. Blair, and Adrian Walker. *Towards a Theory of Declarative Knowledge*, pages 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[8] Andreas Arvanitis, Stephen H. Muggleton, Jianzhong Chen, and Hiroaki Watanabe. Abduction with stochastic logic programs based on a possible worlds semantics. In *Short Paper Proceedings of the 16th International Conference on Inductive Logic Programming (ILP-06)*. University of Coruña, 2006.

[9] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[10] Damiano Azzolini, Elena Bellodi, Alessandro Brancaleoni, Fabrizio Riguzzi, and Evelina Lamma. Modeling bitcoin lightning network by logic programming. *Proceedings 36th International Conference on Logic Programming (Technical Communications)*, 325:258–260, 2020.

[11] Damiano Azzolini, Elena Bellodi, Alessandro Brancaleoni, Fabrizio Riguzzi, and Evelina Lamma. Modeling bitcoin lightning network by logic programming. In Francesco Ricca, Alessandra Russo, Sergio Greco, Nicola Leone, Alexander Artikis, Gerhard Friedrich, Paul Fodor, Angelika Kimmig, Francesca Lisi, Marco Maratea, Alessandra Mileo, and Fabrizio Riguzzi, editors, *Proceedings of the 36th International Conference on Logic Programming (Technical Communications)*, pages 258–260, Waterloo, Australia, 2020. Open Publishing Association.

[12] Damiano Azzolini, Elena Bellodi, Stefano Ferilli, Fabrizio Riguzzi, and Riccardo Zese. Abduction with probabilistic logic programming under the distribution semantics. *International Journal of Approximate Reasoning*, 142:41–63, 2022.

[13] Damiano Azzolini and Fabrizio Riguzzi. Optimizing probabilities in probabilistic logic programs. *Theory and Practice of Logic Programming*, 21(5):543–556, 2021.

[14] Damiano Azzolini and Fabrizio Riguzzi. Reducing probabilistic logic programs. In Ahmet Soylu, Alireza Tamaddoni Nezhad, Nikolay Nikolov, Ioan Toma, Anna Fensel, and Joost Vennekens, editors, *Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium at RuleML+RR 2021 co-located with 17th Reasoning Web Summer School (RW 2021) and 13th DecisionCAMP 2021 as part of Declarative AI 2021*, volume 2956 of *CEUR Workshop Proceedings*, pages 1–13, Aachen, Germany, 2021. Sun SITE Central Europe.

[15] Damiano Azzolini and Fabrizio Riguzzi. Syntactic requirements for well-defined hybrid probabilistic logic programs. In Andrea Formisano, Yanhong Annie Liu, Bart Bogaerts, Alex Brik, Veronica Dahl, Carmine Dodaro, Paul Fodor, Gian Luca Pozzato, Joost Vennekens, and Neng-Fa Zhou, editors, *Proceedings 37th International Conference on Logic Programming (Technical Communications)*, pages 14–26, Waterloo, Australia, 2021. Open Publishing Association.

[16] Damiano Azzolini, Fabrizio Riguzzi, Elena Bellodi, and Evelina Lamma. A probabilistic logic model of lightning network. In *Business Information Systems Workshops*, volume In press of *Springer's Lecture Notes in Business Information Processing (LNBIP)*, Cham, Switzerland, 2021. Springer International Publishing.

[17] Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Analyzing transaction fees with probabilistic logic programming. In Witold Abramowicz and Rafael Corchuelo, editors, *Business Information Systems Workshops*, pages 243–254, Cham, 2019. Springer International Publishing.

[18] Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Studying transaction fees in the bitcoin blockchain with probabilistic logic programming. *Information*, 10(11):335, 2019.

[19] Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. An analysis of Gibbs sampling for probabilistic logic programs. In Carmine Dodaro, George Aristidis Elder, Wolfgang Faber, Jorge Fandinno, Martin Gebser, Markus Hecher, Emily LeBlanc, Michael Morak, and Jessica Zangari,

editors, *Workshop on Probabilistic Logic Programming (PLP 2020)*, volume 2678 of *CEUR-WS*, pages 1–13, Aachen, Germany, 2020. Sun SITE Central Europe.

[20] Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. Modeling smart contracts with probabilistic logic programming. In Witold Abramowicz and Gary Klein, editors, *Business Information Systems Workshops*, volume 394 of *Lecture Notes in Business Information Processing*, pages 86–98, Cham, 2020. Springer International Publishing.

[21] Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. A semantics for hybrid probabilistic logic programs with function symbols. *Artificial Intelligence*, 294:103452, 2021.

[22] Damiano Azzolini, Fabrizio Riguzzi, Evelina Lamma, Elena Bellodi, and Riccardo Zese. Modeling bitcoin protocols with probabilistic logic programming. In Elena Bellodi and Tom Schrijvers, editors, *Proceedings of the 5th International Workshop on Probabilistic Logic Programming, PLP 2018, co-located with the 28th International Conference on Inductive Logic Programming (ILP 2018), Ferrara, Italy, September 1, 2018.*, volume 2219 of *CEUR Workshop Proceedings*, pages 49–61, 2018.

[23] Damiano Azzolini, Fabrizio Riguzzi, Evelina Lamma, and Franco Masotti. A comparison of MCMC sampling for probabilistic logic programming. In Mario Alviano, Gianluigi Greco, and Francesco Scarcello, editors, *Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019*, volume 11946 of *Lecture Notes in Computer Science*, Heidelberg, Germany, 2019. Springer.

[24] Behrouz Babaki, Tias Guns, and Luc de Raedt. Stochastic constraint programming with and-or branch-and-bound. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 539–545, 2017.

[25] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014.

[26] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1):57–144, 2009.

[27] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*, 2017.

[28] Massimo Bartoletti, Barbara Pes, and Sergio Serusi. Data mining for detecting bitcoin ponzi schemes. In *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*, pages 75–84. IEEE, 2018.

[29] Martijn Bastiaan. Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin, 2015. `https://fmt.ewi.utwente.nl/media/175.pdf`, accessed October 20, 2021.

[30] Soumya Basu, David Easley, Maureen O'Hara, and Emin Gün Sirer. Towards a functional fee market for cryptocurrencies. *CoRR*, abs/1901.06830, 2019.

[31] Logan Beal, Daniel Hill, R Martin, and John Hedengren. Gekko optimization suite. *Processes*, 6(8):106, 2018.

[32] Elena Bellodi, Marco Alberti, Fabrizio Riguzzi, and Riccardo Zese. MAP inference for probabilistic logic programming. *Theory and Practice of Logic Programming*, 20(5):641–655, 2020.

[33] Elena Bellodi, Marco Gavanelli, Riccardo Zese, Evelina Lamma, and Fabrizio Riguzzi. Nonground abductive logic programming with probabilistic integrity constraints. *Theory and Practice of Logic Programming*, 21(5):557–574, 2021.

[34] Elena Bellodi and Fabrizio Riguzzi. Learning the structure of probabilistic logic programs. In Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *22nd International Conference on Inductive Logic Programming*, volume 7207 of *LNCS*, pages 61–75. Springer Berlin Heidelberg, 2012.

[35] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(2):169–212, 2015.

[36] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[37] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[38] R. Bowden, H. Paul Keeler, Anthony E. Krzesinski, and Peter G. Taylor. Block arrivals in the bitcoin blockchain. *CoRR*, abs/1801.07447, 2018.

[39] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011.

[40] Vitalik Buterin. A next-generation smart contract and decentralized application platform, 2014. `https://github.com/ethereum/wiki/wiki/White-Paper`, accessed February 14, 2019.

[41] Turliuc Calin-Rares, Maimari Nataly, Russo Alessandra, and Broda Krysia. On minimality and integrity constraints in probabilistic abduction. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 759–775. Springer, 2013.

[42] Miles Carlsten, Harry Kalodner, S Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 154–167. ACM, 2016.

[43] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. Quantitative analysis of smart contracts. In *European Symposium on Programming*, pages 739–767. Springer, Cham, 2018.

[44] Natalia Chaudhry and Muhammad Yousaf. Consensus algorithms in blockchain: Comparative analysis, challenges and opportunities. In *12th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 54–63, 12 2018.

[45] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[46] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[47] Yuan S. Chow and Henry Teicher. *Probability Theory: Independence, Interchangeability, Martingales.* Springer Texts in Statistics. Springer, 2012.

[48] Henning Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In Tom Schrijvers and Thom Frühwirth, editors, *Constraint Handling Rules*, volume 5388 of *Lecture Notes in Computer Science*, pages 85–118. Springer, 2008.

[49] Henning Christiansen and John P. Gallagher. Non-discriminating arguments and their uses. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.

[50] Giovanni Ciatto, Roberta Calegari, Stefano Mariani, Enrico Denti, and Andrea Omicini. From the blockchain to logic programming and back: Research perspectives. In Massimo Cossentino, Luca Sabatucci, and Valeria Seidita, editors, *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018.*, volume 2215 of *CEUR Workshop Proceedings*, pages 69–74. CEUR-WS.org, 2018.

[51] Krzysztof Ciesielski. *Set Theory for the Working Mathematician.* London Mathematical Society Student Texts. Cambridge University Press, 1997.

[52] Keith L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[53] Alain Colmerauer, Henri Kanoui, Robert Pasero, and Philippe Roussel. Un systeme de communication homme-machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[54] Shaen Corbet, Brian Lucey, Andrew Urquhart, and Larisa Yarovaya. Cryptocurrencies as a financial asset: A systematic analysis. *International Review of Financial Analysis*, 62:182–199, 2019.

[55] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[56] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[57] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

[58] Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. Compressing probabilistic Prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.

[59] Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[60] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI*, pages 2462–2467, 2007.

[61] Ittay Eyal and Emin Gün Sirer. How to disincentivize large bitcoin mining pools, 2014.

[62] Stefano Ferilli. Extending expressivity and flexibility of abductive logic programming. *Journal of Intelligent Information Systems*, 51:647–672, 2018.

[63] Daan Fierens, Guy Van den Broeck, Maurice Bruynooghe, and Luc De Raedt. Constraints for probabilistic logic programming. In D. Roy, V. Mansinghka, and N. Goodman, editors, *Proceedings of the NIPS Probabilistic Programming Workshop*, 2012.

[64] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[65] Carsten Fritz. Some fixed point basics. In *Automata Logics, and Infinite Games*, pages 359–364. Springer, 2002.

[66] Tze Ho Fung and Robert A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[67] David Garcia, Claudio J. Tessone, Pavlin Mavrodiev, and Nicolas Perony. The digital traces of bubbles: feedback cycles between socio-economic signals in the bitcoin economy. *Journal of the Royal Society Interface*, 11(99):20140623, 2014.

[68] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, volume 88, pages 1070–1080. MIT Press, 1988.

[69] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In *Readings in computer vision*, pages 564–584. Elsevier, 1987.

[70] Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending problog with continuous distributions. In Paolo Frasconi and Francesca A. Lisi,

editors, *20th International Conference on Inductive Logic Programming (ILP 2010)*, volume 6489 of *LNCS*, pages 76–91. Springer, 2011.

[71] Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter learning in probabilistic databases: A least squares approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008)*, volume 5211 of *LNCS*, pages 473–488. Springer, 2008.

[72] Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2011)*, volume 6911 of *LNCS*, pages 581–596. Springer, 2011.

[73] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 11(4-5):663–680, 2011.

[74] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[75] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008.

[76] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[77] Pascal Hitzler and Anthony Seda. *Mathematical Aspects of Logic Programming Semantics*. Chapman & Hall/CRC Studies in Informatics Series. CRC Press, 2016.

[78] Joe Hurd. A formal approach to probabilistic termination. In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *LNCS*, pages 230–245. Springer, 2002.

[79] Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 167–183. Springer, 2016.

[80] Katsumi Inoue, Taisuke Sato, Masakazu Ishihata, Yoshitaka Kameya, and Hidetomo Nabeshima. Evaluating abductive hypotheses using an EM algorithm on BDDs. In *21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 810–815. Morgan Kaufmann Publishers Inc., 2009.

[81] Muhammad Asiful Islam, CR Ramakrishnan, and IV Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12:505–523, 2012.

[82] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.

[83] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[84] Chuan Jiang, Junaid Babar, Gianfranco Ciardo, Andrew S Miner, and Benjamin Smith. Variable reordering in binary decision diagrams. In *26th International Workshop on Logic and Synthesis*, pages 1–8, 2017.

[85] Steven G. Johnson. The nlopt nonlinear-optimization package, 2020.

[86] Antonis C. Kakas, R. A. Kowalski, and Francesca Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[87] Antonis C. Kakas and Paolo Mancarella. Abductive logic programming. In *Proceedings of NACLP Workshop on Non-Monotonic Reasoning and Logic Programming*, 1990.

[88] Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In *Proceedings of the 16th VLDB*, pages 650–661. Morgan Kaufmann, 1990.

[89] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[90] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In Peter Thiemann, editor, *25th European Symposium on Programming, on Programming Languages and Systems (ESOP 2016)*, volume 9632 of *LNCS*, pages 364–389. Springer, 2016.

[91] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive*, 2012(248), 2012.

[92] Shoji Kasahara and Jun Kawahara. Priority mechanism of bitcoin and its effect on transaction-confirmation process. *CoRR*, abs/1604.00103, 2016.

[93] Rohit J. Kate and Raymond J. Mooney. Probabilistic abduction using markov logic networks. In *Proceedings of the IJCAI-09 Workshop on Plan, Activity, and Intent Recognition (PAIR-09)*, Pasadena, CA, July 2009.

[94] Paraskevi Katsiampa. Volatility estimation for bitcoin: A comparison of garch models. *Economics Letters*, 158:3–6, 2017.

[95] Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. On the efficient execution of ProbLog programs.

In *24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 175–189. Springer, December 2008.

[96] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, volume 1, pages 209–214. AAAI Press, 2011.

[97] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, Cambridge, MA, 2009.

[98] David T. Koops. Predicting the confirmation time of bitcoin transactions. *CoRR*, abs/1809.10596, 2018.

[99] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[100] Dieter Kraft. Algorithm 733: Tomp-fortran modules for optimal control calculations. *ACM Trans. Math. Softw.*, 20(3):262–281, September 1994.

[101] Ladislav Kristoufek. Bitcoin meets google trends and wikipedia: Quantifying the relationship between phenomena of the internet era. *Scientific reports*, 3:3415, 2013.

[102] Anna L. D. Latour, Behrouz Babaki, Anton Dries, Angelika Kimmig, Guy Van den Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and probabilistic programming. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 495–511, Cham, 2017. Springer International Publishing.

[103] Joohyung Lee, Samidh Talsania, and Yi Wang. Computing lpmln using asp and mln solvers. *Theory and Practice of Logic Programming*, 17(5-6):942–960, 2017.

[104] Kevin Liao and Jonathan Katz. Incentivizing blockchain forks via whale transactions. In *International Conference on Financial Cryptography and Data Security*, pages 264–279. Springer, 2017.

[105] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[106] Michele Lombardi and Michela Milano. Allocation and scheduling of conditional task graphs. *Artificial Intelligence*, 174(7):500–529, 2010.

[107] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[108] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 17–30, New York, NY, USA, 2016. ACM.

[109] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, Sep 2019.

[110] Wannes Meert, Jan Struyf, and Hendrik Blockeel. Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, 89(1):131–160, 2008.

[111] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[112] Steffen Michels, Arjen Hommersom, Peter J. F. Lucas, and Marina Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence*, 228:1–44, 2015.

[113] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *International Conference on Financial Cryptography and Data Security*, pages 19–33. Springer, 2015.

[114] Stephen Muggleton. Learning stochastic logic programs. In Lise Getoor and David Jensen, editors, *Learning Statistical Models from Relational Data, Papers from the 2000 AAAI Workshop*, volume WS-00-06 of *AAAI Workshops*, pages 36–41. AAAI Press, 2000.

[115] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[116] Stephen Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.

[117] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `https://bitcoin.org/bitcoin.pdf`, accessed October 20, 2021.

[118] Arun Nampally and CR Ramakrishnan. Adaptive MCMC-based inference in probabilistic logic programs. *arXiv preprint arXiv:1403.6036*, 2014.

[119] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[120] Matthias Nickles. A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics. In Loizos Michael and Antonis Kakas, editors, *Logics in Artificial Intelligence*, pages 369–384, Cham, 2016. Springer International Publishing.

[121] Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. *Machine Learning*, 103(3):407–449, 2016.

[122] Francesco Orsini, Paolo Frasconi, and Luc De Raedt. kProbLog: an algebraic prolog for machine learning. *Machine Learning*, 106(12):1933–1969, 2017.

[123] Dmytro Piatkivskyi and Mariusz Nowostawski. Split payments in payment networks. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 67–75. Springer, 2018.

[124] Carlos Pinzón and Camilo Rocha. Double-spend attack models with time advantange for bitcoin. *Electr. Notes Theor. Comput. Sci.*, 329:79–103, 2016.

[125] David 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.

[126] David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[127] David Poole. Abducing through negation as failure: Stable models within the independent choice logic. *Journal of Logic Programming*, 44(1–3):5–35, 2000.

[128] David Poole. The independent choice logic and beyond. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, pages 222–243. Springer, 2008.

[129] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[130] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[131] Teodor C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-1989)*, pages 11–21. ACM Press, 1989.

[132] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[133] Fabrizio Riguzzi. Extended semantics and inference for the independent choice logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009.

[134] Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013.

[135] Fabrizio Riguzzi. The distribution semantics for normal programs with function symbols. *International Journal of Approximate Reasoning*, 77:1–19, 2016.

[136] Fabrizio Riguzzi. *Foundations of Probabilistic Logic Programming: Languages, semantics, inference and learning.* River Publishers, Gistrup, Denmark, 2018.

[137] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Marco Alberti, and Evelina Lamma. Probabilistic inductive constraint logic. *Machine Learning*, 110:1–32, 04 2021.

[138] Fabrizio Riguzzi and Nicola Di Mauro. Applying the information bottleneck to statistical relational learning. *Machine Learning*, 86(1):89–114, 2012.

[139] Fabrizio Riguzzi and Terrance Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010)*, volume 7 of *LIPIcs*, pages 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[140] Fabrizio Riguzzi and Terrance Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2):279–302, 2013.

[141] Elias Rohrer, Julian Malliaris, and Florian Tschorsch. Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 347–356. IEEE, 2019.

[142] Meni Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.

[143] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4292–4293. AAAI Press, 2015.

[144] Tian Sang, Paul Bearne, and Henry Kautz. Performing bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, pages 475–481. AAAI Press, 2005.

[145] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[146] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 715–729. MIT Press, 1995.

[147] Taisuke Sato. EM learning for symbolic-statistical models in statistical abduction. In *Progress in Discovery Science, Final Report of the Japanese Discovery Science Project*, pages 189–200. Springer, 2002.

[148] Taisuke Sato and Yoshitaka Kameya. PRISM: a language for symbolic-statistical modeling. In *15th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, volume 97, pages 1330–1339, 1997.

[149] Taisuke Sato and Yoshitaka Kameya. A viterbi-like algorithm and em learning for statistical abduction. In *Proceedings of UAI2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support*, 2000.

[150] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[151] Taisuke Sato and Philipp Meyer. Tabling for infinite probability computation. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPIcs*, pages 348–358. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[152] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[153] István Seres, László Gulyás, Dániel Nagy, and Péter Burcsi. *Topological Analysis of Bitcoin's Lightning Network*, pages 1–12. Springer International Publishing, 01 2020.

[154] Dimitar Sht. Shterionov, Joris Renkens, Jonas Vlasselaer, Angelika Kimmig, Wannes Meert, and Gerda Janssens. The most probable explanation for probabilistic logic programs with annotated disjunctions. In Jesse Davis and Jan Ramon, editors, *24th International Conference on Inductive Logic Programming (ILP 2014)*, volume 9046 of *LNCS*, pages 139–153, Berlin, Heidelberg, 2015. Springer.

[155] Gerardo Simari and V. S. Subrahmanian. Abductive Inference in Probabilistic Logic Programs. In Manuel Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 192–201, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[156] Gerardo I. Simari, John P. Dickerson, Amy Sliva, and V. S. Subrahmanian. Parallel abductive query answering in probabilistic logic programs. *ACM Trans. Comput. Logic*, 14(2), June 2013.

[157] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 3.0.0*. University of Colorado, 2015.

[158] Krister Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM Journal on Optimization*, pages 555–573, 2002.

[159] Terrance Swift and David Scott Warren. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[160] Nick Szabo. Smart contracts, 1994. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[161] Itay Tsabary and Ittay Eyal. The gap game. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 713–728. ACM, 2018.

[162] Endriss Ulrich, Mancarella Paolo, Sadri Fariba, Terreni Giacomo, and Toni Francesca. Abductive logic programming with CIFF: System description. In J.J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence. JELIA 2004*, volume 3229 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2004. Springer.

[163] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[164] Guy Van den Broeck, Ingo Thon, Martijn van Otterlo, and Luc De Raedt. DTProbLog: A decision-theoretic probabilistic Prolog. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1217–1222. AAAI Press, 2010.

[165] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[166] Wiebe Van Ranst and Joost Vennekens. An opencl implementation of a forward sampling algorithm for cp-logic. *International Journal of Approximate Reasoning*, 67:60–72, 2015.

[167] Marie Vasek and Tyler Moore. Analyzing the bitcoin ponzi scheme ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 101–112. Springer, 2018.

[168] Joost Vennekens, Marc Denecker, and Maurice 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.

[169] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In Bart Demoen and Vladimir Lifschitz, editors, *20th International Conference on Logic Programming (ICLP 2004)*, volume 3131 of *LNCS*, pages 431–445. Springer, 2004.

[170] Toby Walsh. Stochastic constraint programming. *Proceedings of the 15th European Conference on Artificial Intelligence*, 1:111–115, 2002.

[171] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[172] Mark T. Williams. Virtual currencies–bitcoin risk, 2014.

[173] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[174] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Knowledge compilation with continuous random variables and its application in hybrid probabilistic logic programming. *CoRR*, abs/1807.00614, 2018.