

River Publishers Series in Software Engineering

FOUNDATIONS OF PROBABILISTIC LOGIC PROGRAMMING

Languages, Semantics,
Inference and Learning

Fabrizio Riguzzi



River Publishers

**Foundations of Probabilistic
Logic Programming**
Languages, Semantics,
Inference and Learning

RIVER PUBLISHERS SERIES IN SOFTWARE ENGINEERING

Indexing: All books published in this series are submitted to the Web of Science Book Citation Index (BkCI), to CrossRef and to Google Scholar.

The “River Publishers Series in Software Engineering” is a series of comprehensive academic and professional books which focus on the theory and applications of Computer Science in general, and more specifically Programming Languages, Software Development and Software Engineering.

Books published in the series include research monographs, edited volumes, handbooks and textbooks. The books provide professionals, researchers, educators, and advanced students in the field with an invaluable insight into the latest research and developments.

Topics covered in the series include, but are by no means restricted to the following:

- Software Engineering
- Software Development
- Programming Languages
- Computer Science
- Automation Engineering
- Research Informatics
- Information Modelling
- Software Maintenance

For a list of other books in this series, visit www.riverpublishers.com

Foundations of Probabilistic Logic Programming

Languages, Semantics, Inference and Learning

Fabrizio Riguzzi

University of Ferrara
Italy



River Publishers

Published, sold and distributed by:

River Publishers
Alsbjergvej 10
9260 Gistrup
Denmark

River Publishers
Lange Geer 44
2611 PW Delft
The Netherlands

Tel.: +45369953197
www.riverpublishers.com

ISBN: 978-87-7022-018-7 (Hardback)
978-87-7022-017-0 (Ebook)

©2018 River Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Cover image Copyright by Giorgio Morara

Contents

Foreword	xi
Preface	xiii
Acknowledgements	xv
List of Figures	xvii
List of Tables	xxi
List of Examples	xxiii
List of Definitions	xxvii
List of Theorems	xxix
List of Abbreviations	xxxii
1 Preliminaries	1
1.1 Orders, Lattices, Ordinals	1
1.2 Mappings and Fixpoints	3
1.3 Logic Programming	4
1.4 Semantics for Normal Logic Programs	13
1.4.1 Program Completion	13
1.4.2 Well-Founded Semantics	15
1.4.3 Stable Model Semantics	21
1.5 Probability Theory	23
1.6 Probabilistic Graphical Models	32

2	Probabilistic Logic Programming Languages	41
2.1	Languages with the Distribution Semantics	41
2.1.1	Logic Programs with Annotated Disjunctions	42
2.1.2	ProbLog	43
2.1.3	Probabilistic Horn Abduction	43
2.1.4	PRISM	44
2.2	The Distribution Semantics for Programs Without Function Symbols	45
2.3	Examples of Programs	50
2.4	Equivalence of Expressive Power	56
2.5	Translation to Bayesian Networks	58
2.6	Generality of the Distribution Semantics	62
2.7	Extensions of the Distribution Semantics	64
2.8	CP-Logic	66
2.9	Semantics for Non-Sound Programs	71
2.10	KBMC Probabilistic Logic Programming Languages	76
2.10.1	Bayesian Logic Programs	76
2.10.2	CLP(BN)	76
2.10.3	The Prolog Factor Language	79
2.11	Other Semantics for Probabilistic Logic Programming	80
2.11.1	Stochastic Logic Programs	81
2.11.2	ProPPR	82
2.12	Other Semantics for Probabilistic Logics	84
2.12.1	Nilsson’s Probabilistic Logic	84
2.12.2	Markov Logic Networks	84
	2.12.2.1 Encoding Markov Logic Networks with Probabilistic Logic Programming	85
2.12.3	Annotated Probabilistic Logic Programs	88
3	Semantics with Function Symbols	91
3.1	The Distribution Semantics for Programs with Function Symbols	92
3.2	Infinite Covering Set of Explanations	97
3.3	Comparison with Sato and Kameya’s Definition	110
4	Semantics for Hybrid Programs	115
4.1	Hybrid ProbLog	115
4.2	Distributional Clauses	118
4.3	Extended PRISM	124

4.4	cplint Hybrid Programs	126
4.5	Probabilistic Constraint Logic Programming	130
4.5.1	Dealing with Imprecise Probability Distributions	135
5	Exact Inference	145
5.1	PRISM	146
5.2	Knowledge Compilation	150
5.3	ProbLog1	151
5.4	cplint	155
5.5	SLGAD	157
5.6	PITA	158
5.7	ProbLog2	163
5.8	T_P Compilation	176
5.9	Modeling Assumptions in PITA	178
5.9.1	PITA(OPT)	181
5.9.2	MPE with PITA	186
5.10	Inference for Queries with an Infinite Number of Explanations	186
5.11	Inference for Hybrid Programs	187
6	Lifted Inference	195
6.1	Preliminaries on Lifted Inference	195
6.1.1	Variable Elimination	197
6.1.2	GC-FOVE	201
6.2	LP ²	202
6.2.1	Translating ProbLog into PFL	202
6.3	Lifted Inference with Aggregation Parfactors	205
6.4	Weighted First-Order Model Counting	207
6.5	Cyclic Logic Programs	210
6.6	Comparison of the Approaches	210
7	Approximate Inference	213
7.1	ProbLog1	213
7.1.1	Iterative Deepening	213
7.1.2	k -best	215
7.1.3	Monte Carlo	216
7.2	MCINTYRE	218
7.3	Approximate Inference for Queries with an Infinite Number of Explanations	221

7.4	Conditional Approximate Inference	222
7.5	Approximate Inference by Sampling for Hybrid Programs	223
7.6	Approximate Inference with Bounded Error for Hybrid Programs	226
7.7	k -Optimal	229
7.8	Explanation-Based Approximate Weighted Model Counting	231
7.9	Approximate Inference with T_P -compilation	233
7.10	DISTR and EXP Tasks	234
8	Non-Standard Inference	239
8.1	Possibilistic Logic Programming	239
8.2	Decision-Theoretic ProbLog	241
8.3	Algebraic ProbLog	250
9	Parameter Learning	259
9.1	PRISM Parameter Learning	259
9.2	LLPAD and ALLPAD Parameter Learning	265
9.3	LeProbLog	267
9.4	EMBLEM	270
9.5	ProbLog2 Parameter Learning	280
9.6	Parameter Learning for Hybrid Programs	282
10	Structure Learning	283
10.1	Inductive Logic Programming	283
10.2	LLPAD and ALLPAD Structure Learning	287
10.3	ProbLog Theory Compression	289
10.4	ProbFOIL and ProbFOIL+	290
10.5	SLIPCOVER	296
10.5.1	The Language Bias	296
10.5.2	Description of the Algorithm	296
10.5.2.1	Function INITIALBEAMS	298
10.5.2.2	Beam Search with Clause Refinements	300
10.5.3	Execution Example	301
10.6	Examples of Datasets	304
11	cplint Examples	305
11.1	cplint Commands	305
11.2	Natural Language Processing	309
11.2.1	Probabilistic Context-Free Grammars	309

11.2.2	Probabilistic Left Corner Grammars	310
11.2.3	Hidden Markov Models	311
11.3	Drawing Binary Decision Diagrams	313
11.4	Gaussian Processes	314
11.5	Dirichlet Processes	318
11.5.1	The Stick-Breaking Process	319
11.5.2	The Chinese Restaurant Process	322
11.5.3	Mixture Model	324
11.6	Bayesian Estimation	326
11.7	Kalman Filter	327
11.8	Stochastic Logic Programs	330
11.9	Tile Map Generation	332
11.10	Markov Logic Networks	334
11.11	Trael	335
11.12	Coupon Collector Problem	339
11.13	One-Dimensional Random Walk	341
11.14	Latent Dirichlet Allocation	342
11.15	The Indian GPA Problem	346
11.16	Bongard Problems	348
12	Conclusions	351
	References	353
	Index	375
	About the Author	387

Foreword

The computational foundations of Artificial Intelligence (AI) are supported by two corner stones: logics and machine learning. Computational logic has found its realization in a number of frameworks for logic-based approaches to knowledge representation and automated reasoning, such as Logic Programming, Answer Set Programming, Constraint Logic Programming, Description Logics, and Temporal Logics. Machine Learning, and its recent evolution to Deep Learning, has a huge number of applications in video surveillance, social media services, big data analysis, weather predictions, spam filtering, online customer support, etc.

Emerging interest in the two communities for finding a bridge connecting them is witnessed, for instance, by the prize *test-of-time, 20 years* assigned by the association for logic programming in 2017 to the paper *Hybrid Probabilistic Programs*. Also in 2017, Holger H. Hoos was invited to give the talk *The best of both worlds: Machine learning meets logical reasoning* at the international conference on logic programming. Here, machine learning is used to tune the search heuristics in solving combinatorial problems (e.g., encoded using SAT or ASP techniques). A couple of months later, in a panel organized by the Italian Association for Artificial Intelligence (AI*IA), the machine learning researcher Marco Gori posed five questions to the communities. Among them: *How can we integrate huge knowledge bases naturally and effectively with learning processes? How to break the barriers of machine learning vs (inductive) logic programming communities? How to derive a computational model capable of dealing with learning and reasoning both in the symbolic and sub-symbolic domains? How to acquire latent semantics?* These are fundamental questions that need to be resolved to allow AI research to make another quantum leap. Logical languages can add structural semantics to statistical inference.

This book, based on 15 years of top-level research in the field by Fabrizio Riguzzi and his co-authors, addresses these questions and fills most of the gaps between the two communities. A mature, uniform retrospective of several proposals of languages for Probabilistic Logic Programming is reported.

The reader can decide whether to explore all the technical details or simply use such languages without the need of installing tools, by simply using the web site maintained by Fabrizio's group in Ferrara.

The book is self-contained: all the prerequisites coming from discrete mathematics (often at the foundation of logical reasoning) and continuous mathematics, probability, and statistics (at the foundation of machine learning) are presented in detail. Although all proposals are summarized, those based on the distribution semantics are dealt with in a greater level of detail. The book explains how a system can reason precisely or approximately when the size of the program (and data) increases, even in the case on non-standard inference (e.g., possibilistic reasoning). The book then moves toward parameter learning and structure learning, thus reducing and possibly removing the distance with respect to machine learning. The book closes with a lovely chapter with several encodings in PLP. A reader with some knowledge of logic programming can start from this chapter, having fun testing the programs (for instance, discovering the best strategy to be applied during a *truel*, namely, a duel involving three gunners shooting sequentially) and then move to the theoretical part.

As the president of the Italian Association for Logic Programming (GULP) I am proud that this significant effort has been made by one of our associates and former member of our Executive Committee. I believe that it will become a reference book for the new generations that have to deal with the new challenges coming from the need of reasoning on Big Data.

Agostino Dovier
University of Udine

Preface

The field of Probabilistic Logic Programming (PLP) was started in the early 1990s by seminal works such as those of [Dantsin, 1991], [Ng and Subrahmanian, 1992], [Poole, 1993b], and [Sato, 1995].

However, the problem of combining logic and probability has been studied since the 1950s [Carnap, 1950; Gaifman, 1964]. Then the problem became prominent in the field of Artificial Intelligence in the late 1980s to early 1990s when researchers tried to reconcile the probabilistic and logical approaches to AI [Nilsson, 1986; Halpern, 1990; Fagin and Halpern, 1994; Halpern, 2003].

The integration of logic and probability combines the capability of the first to represent complex relations among entities with the capability of the latter to model uncertainty over attributes and relations. Logic programming provides a Turing complete language based on logic and thus represents an excellent candidate for the integration.

Since its birth, the field of Probabilistic Logic Programming has seen a steady increase of activity, with many proposals for languages and algorithms for inference and learning. The language proposals can be grouped into two classes: those that use a variant of the Distribution Semantics (DS) [Sato, 1995] and those that follow a Knowledge Base Model Construction (KBMC) approach [Wellman et al., 1992; Bacchus, 1993].

Under the DS, a probabilistic logic program defines a probability distribution over normal logic programs and the probability of a ground query is then obtained from the joint distribution of the query and the programs. Some of the languages following the DS are: Probabilistic Logic Programs [Dantsin, 1991], Probabilistic Horn Abduction [Poole, 1993b], PRISM [Sato, 1995], Independent Choice Logic [Poole, 1997], pD [Fuhr, 2000], Logic Programs with Annotated Disjunctions [Vennekens et al., 2004], ProbLog [De Raedt et al., 2007], P-log [Baral et al., 2009], and CP-logic [Vennekens et al., 2009].

Instead, in KBMC languages, a program is seen as a template for generating a ground graphical model, be it a Bayesian network or a Markov network. KBMC languages include Relational Bayesian Network

[Jaeger, 1998], CLP(BN) [Costa et al., 2003], Bayesian Logic Programs [Kersting and De Raedt, 2001], and the Prolog Factor Language [Gomes and Costa, 2012]. The distinction among DS and KBMC languages is actually non-sharp as programs in languages following the DS can also be translated into graphical models.

This book aims at providing an overview of the field of PLP, with a special emphasis on languages under the DS. The reason is that their approach to logic-probability integration is particularly simple and coherent across languages but nevertheless powerful enough to be useful in a variety of domains. Moreover, they can be given a semantics in purely logical terms, without necessarily resorting to a translation into graphical models.

The book doesn't aim though at being a complete account of the topic, even when restricted to the DS, as the field has grown large, with a dedicated workshop series started in 2014. My objective is to present the main ideas for semantics, inference, and learning and to highlight connections between the methods.

The intended audience of the book are researchers in Computer Science and AI that want to get an overview of PLP. However, it can also be used by students, especially graduate, to get acquainted with the topic, and by practitioners that would like to get more details on the inner workings of methods.

Many examples of the book include a link to a page of the web application `cplint` on SWISH (<http://cplint.eu>) [Riguzzi et al., 2016a; Alberti et al., 2017], where the code can be run online using `cplint`, a system we developed at the University of Ferrara that includes many algorithms for inference and learning in a variety of languages.

The book starts with Chapter 1 that presents preliminary notions of logic programming and graphical models. Chapter 2 introduces the languages under the DS, discusses the basic form of the semantics, and compares it with alternative approaches in PLP and AI in general. Chapters 3 and 4 describe the semantics for more complex cases, the first of languages allowing function symbols and the latter allowing continuous random variables. Chapter 5 presents various algorithms for exact inference. Lifted inference is discussed in Chapter 6 and approximate inference in Chapter 7. Non-standard inference problems are illustrated in Chapter 8. Then Chapters 9 and 10 treat the problem of learning parameters and structure of programs, respectively. Chapter 11 presents some examples of use of the system `cplint`. Chapter 12 concludes the book discussing open problems.

Acknowledgments

I am indebted to many persons for their help and encouragement. Evelina Lamma and Paola Mello taught me to love logical reasoning and always supported me, especially during the bad times. My co-workers at the University of Ferrara Evelina Lamma, Elena Bellodi, Riccardo Zese, Giuseppe Cota, Marco Alberti, Marco Gavanelli, and Arnaud Nguembang Fadja greatly helped me shape my view of PLP through exiting joint work and insightful discussions. I have been lucky enough to collaborate also with Theresa Swift, Nicola Di Mauro, Stefano Bragaglia, Vitor Santos Costa, and Jan Wielemaker and the joint work with them has found its way into the book.

Agostino Dovier, Evelina Lamma, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Marco Alberti read drafts of the book and gave me very useful comments.

I would also like to thank Michela Milano, Federico Chesani, Paolo Torroni, Luc De Raedt, Angelika Kimmig, Wannes Meert, Joost Vennekens, and Kristian Kersting for many enlightening exchanges of ideas.

This book evolved from a number of articles. In particular, Chapter 2 is based on [Riguzzi and Swift, 2018], Chapter 3 on [Riguzzi, 2016], Section 5.6 on [Riguzzi and Swift, 2010, 2011, 2013], Section 5.9 on [Riguzzi, 2014], Section 7.2 on [Riguzzi, 2013], Chapter 6 on [Riguzzi et al., 2017a], Section 9.4 on [Bellodi and Riguzzi, 2013, 2012], Section 10.2 on [Riguzzi, 2004, 2007b, 2008b], Section 10.5 on [Bellodi and Riguzzi, 2015], and Chapter 11 on [Riguzzi et al., 2016a; Alberti et al., 2017; Riguzzi et al., 2017b; Nguembang Fadja and Riguzzi, 2017].

Finally, I would like to thank my wife Cristina for putting up with a husband with the crazy idea of writing a book without taking a sabbatical. Without her love and support, I would not have been able to bring the idea into reality.

List of Figures

Figure 1.1	SLD tree for the query $path(a, c)$ from the program of Example 1.	10
Figure 1.2	SLDNF tree for the query $ends(b, c)$ from the program of Example 1.	11
Figure 1.3	SLDNF tree for the query c from the program of Example 2.	16
Figure 1.4	Ordinal powers of IFP^P for the program of Example 4.	20
Figure 1.5	Gaussian densities.	28
Figure 1.6	Bivariate Gaussian density.	29
Figure 1.7	Example of a Bayesian network.	35
Figure 1.8	Markov blanket.	36
Figure 1.9	Example of a Markov network.	38
Figure 1.10	Bayesian network equivalent to the Markov network of Figure 1.9.	39
Figure 1.11	Example of a factor graph.	40
Figure 2.1	Example of a BN.	56
Figure 2.2	BN $\beta(\mathcal{P})$ equivalent to the program of Example 28.	60
Figure 2.3	Portion of $\gamma(\mathcal{P})$ relative to a clause C_I	61
Figure 2.4	BN $\gamma(\mathcal{P})$ equivalent to the program of Example 28.	62
Figure 2.5	BN representing the dependency between $a(i)$ and $b(i)$	63
Figure 2.6	BN modeling the distribution over $a(i), b(i), X_1, X_2, X_3$	64
Figure 2.7	Probability tree for Example 2.11.	67
Figure 2.8	An incorrect probability tree for Example 30.	68
Figure 2.9	A probability tree for Example 30.	68
Figure 2.10	Ground Markov network for the MLN of Example 39.	85
Figure 4.1	Credal set specification for Examples 62 and 64.	137

Figure 5.1	Explanations for query $hmm([a, b, b])$ of Example 65.	148
Figure 5.2	PRISM formulas for query $hmm([a, b, b])$ of Example 65.	149
Figure 5.3	PRISM computations for query $hmm([a, b, b])$ of Example 65.	150
Figure 5.4	BDD representing Function 5.1.	153
Figure 5.5	BDD for query <i>epidemic</i> of Example 66.	154
Figure 5.6	MDD for the diagnosis program of Example 19.	156
Figure 5.7	BDD representing the function in Equation (5.2).	161
Figure 5.8	d-DNNF for the formula of Example 72.	168
Figure 5.9	Arithmetic circuit for the d-DNNF of Figure 5.8.	169
Figure 5.10	BDD for the formula of Example 72.	172
Figure 5.11	SDD for the formula of Example 72.	173
Figure 5.12	vtree for which the SDD of Figure 5.11 is normalized.	173
Figure 5.13	Examples of graphs satisfying some of the assumptions.	180
Figure 5.14	Code for the <i>or/3</i> predicate of PITA(OPT).	182
Figure 5.15	Code for the <i>and/3</i> predicate of PITA(OPT).	183
Figure 5.16	Code for the <i>exc/2</i> predicate of PITA(OPT).	184
Figure 5.17	Code for the <i>ind/2</i> predicate of PITA(OPT).	185
Figure 5.18	Code for the <i>ev/2</i> predicate of PITA(OPT).	185
Figure 6.1	BN representing an OR dependency between X and Y.	198
Figure 6.2	BN representing a noisy-OR dependency between X and Y.	199
Figure 6.3	BN of Figure 6.1 after deputation.	201
Figure 7.1	Program for Example 85.	214
Figure 7.2	Probabilistic graph of Example 85.	214
Figure 7.3	SLD tree up to depth 4 for the query $path(c, d)$ from the program of Example 85.	214
Figure 7.4	HPT for Example 88 (from [Michels et al., 2016]).	227
Figure 7.5	PHPT for Example 89.	228
Figure 7.6	Distribution of sampled values in the Program of Example 90.	237
Figure 7.7	Distribution of sampled values from the Gaussian mixture of Example 91.	237
Figure 8.1	BDD $_{dry}(\sigma)$ for Example 93.	244

Figure 8.2	BDD _{<i>broken_umbrella</i>} (σ) for Example 93.	244
Figure 8.3	ADD(<i>dry</i>) for Example 93. The dashed terminals indicate ADD ^{util} (<i>dry</i>).	248
Figure 8.4	ADD(<i>broken_umbrella</i>) for Example 93. The dashed terminals indicate ADD ^{util} (<i>broken_umbrella</i>).	248
Figure 8.5	ADD _{tot} ^{util} for Example 93.	248
Figure 8.6	Worlds where the query <i>calls(mary)</i> from Example 97 is true.	253
Figure 9.1	BDD for query <i>epidemic</i> for Example 99.	272
Figure 9.2	BDD after applying the merge rule only for Example 99.	273
Figure 9.3	Forward and backward probabilities. <i>F</i> indicates the forward probability and <i>B</i> the backward probability of each node.	279
Figure 11.1	BDD for query <i>pandemic</i> in the <i>epidemic.pl</i> example, drawn using the CUDD function for exporting the BDD to the dot format of Graphviz.	314
Figure 11.2	Functions sampled from a Gaussian process with a squared exponential kernel in <i>gpr.pl</i>	318
Figure 11.3	Functions from a Gaussian process predicting points with $X = [0, \dots, 10]$ with a squared exponential kernel in <i>gpr.pl</i>	318
Figure 11.4	Distribution of indexes with concentration parameter 10 for the stick-breaking example <i>dirichlet_process.pl</i>	321
Figure 11.5	Distribution of values with concentration parameter 10 for the stick-breaking example <i>dirichlet_process.pl</i>	321
Figure 11.6	Distribution of unique indexes with concentration parameter 10 for the stick-breaking example <i>dirichlet_process.pl</i>	322
Figure 11.7	Prior density in the <i>dp_mix.pl</i> example.	325
Figure 11.8	Posterior density in the <i>dp_mix.pl</i> example.	325
Figure 11.9	Prior and posterior densities in <i>gauss_mean_est.pl</i>	326
Figure 11.10	Prior and posterior densities in <i>kalman.pl</i>	328
Figure 11.11	Example of particle filtering in <i>kalman.pl</i>	329
Figure 11.12	Particle filtering for a 2D Kalman filter.	330

Figure 11.13	Samples of sentences of the language defined in <code>slp_pcfg.pl</code>	331
Figure 11.14	A random tile map.	333
Figure 11.15	Probability tree of the truel with opponents a and b	336
Figure 11.16	Distribution of the number of boxes.	340
Figure 11.17	Expected number of boxes as a function of the number of coupons.	341
Figure 11.18	Smoothed LDA.	343
Figure 11.19	Values for word in position 1 of document 1.	345
Figure 11.20	Values for couples (word,topic) in position 1 of document 1.	345
Figure 11.21	Prior distribution of topics for word in position 1 of document 1.	346
Figure 11.22	Posterior distribution of topics for word in position 1 of document 1.	346
Figure 11.23	Density of the probability of topic 1 before and after observing that words 1 and 2 of document 1 are equal.	347
Figure 11.24	Bongard pictures.	349

List of Tables

Table 2.1	Conditional probability table for a_2	59
Table 2.2	Conditional probability table for ch_5	59
Table 5.1	Tractability of operations. ? means “unknown”, \checkmark means “tractable” and \circ means “not tractable unless P=NP” Operations are meant over a bounded number of operands and BDDs operands should have the same variable order and SDDs the same vtree. From [Vlasselaer et al., 2014]	175
Table 8.1	Inference tasks and corresponding semirings for aProbLog	252
Table 11.1	Associations between variable indexes and ground rules	314

List of Examples

1	Example (Path – Prolog)	9
2	Example (Clark’s completion)	15
3	Example (WFS computation)	18
4	Example (Fixpoint of IFP^P beyond ω)	19
5	Example (Answer set computation)	22
6	Example (Probability spaces)	24
7	Example (Discrete random variables)	25
8	Example (Continuous random variables)	27
9	Example (Joint distributions)	28
10	Example (Alarm – BN)	35
11	Example (University – MN)	38
12	Example (Medical symptoms – LPAD)	42
13	Example (Medical symptoms – ProbLog)	43
14	Example (Medical symptoms – ICL)	44
15	Example (Coin tosses – PRISM)	45
16	Example (Medical symptoms – PRISM)	45
17	Example (Medical symptoms – worlds – ProbLog)	48
18	Example (Medical symptoms – worlds – LPAD)	50
19	Example (Detailed medical symptoms – LPAD)	51
20	Example (Coin – LPAD)	51
21	Example (Eruption – LPAD)	51
22	Example (Monty Hall puzzle – LPAD)	52
23	Example (Three-prisoner puzzle – LPAD)	53
24	Example (Russian roulette with two guns – LPAD)	54
25	Example (Mendelian rules of inheritance – LPAD)	54
26	Example (Path probability – LPAD)	55
27	Example (Alarm BN – LPAD)	55
28	Example (LPAD to BN)	59
29	Example (CP-logic program – infection [Vennekens et al., 2009])	66

30	Example (CP-logic program – pneumonia [Vennekens et al., 2009])	67
31	Example (Invalid CP-logic program [Vennekens et al., 2009])	70
32	Example (Sound LPAD – invalid CP-theory Vennekens et al. [2009])	71
33	Example (Insomnia [Cozman and Mauá, 2017])	72
34	Example (Insomnia – continued – [Cozman and Mauá, 2017])	73
35	Example (Barber paradox – [Cozman and Mauá, 2017])	74
36	Example (PFL program)	80
37	Example (Probabilistic context-free grammar – SLP)	82
38	Example (ProPPR program)	82
39	Example (Markov Logic Network)	85
40	Example (Program with infinite set of worlds)	91
41	Example (Game of dice)	92
42	Example (Covering set of explanations for Example 40)	96
43	Example (Covering set of explanations for Example 41)	96
44	Example (Pairwise incompatible covering set of explanations for Example 40)	97
45	Example (Pairwise incompatible covering set of explanations for Example 41)	97
46	Example (Probability of the query for Example 40)	101
47	Example (Probability of the query for Example 41)	102
48	Example (Gaussian mixture – Hybrid ProbLog)	115
49	Example (Query over a Gaussian mixture – Hybrid ProbLog)	117
50	Example (Gaussian mixture – DCs)	119
51	Example (Moving people – DCs [Nitti et al., 2016])	119
52	Example ($ST_{(P)}$ for moving people – DC [Nitti et al., 2016])	122
53	Example ($ST_{(P)}$ for moving people – DC [Nitti et al., 2016])	123
54	Example (Negation in DCs [Nitti et al., 2016])	124
55	Example (Gaussian mixture – Extended PRISM)	125
56	Example (Gaussian mixture and constraints – Extended PRISM)	125
57	Example (Gaussian mixture – cplint)	127
58	Example (Estimation of the mean of a Gaussian – cplint)	127
59	Example (Kalman filter – cplint)	128

60	Example (Fire on a ship [Michels et al., 2015])	131
61	Example (Probability of fire on a ship [Michels et al., 2015])	133
62	Example (Credal set specification – continuous variables) .	136
63	Example (Credal set specification – discrete variables) . . .	138
64	Example (Conditional probability bounds)	139
65	Example (Hidden Markov model – PRISM [Sato and Kameya, 2008])	147
66	Example (Epidemic – ProbLog)	153
67	Example (Detailed medical symptoms – MDD)	156
68	Example (Medical example – PITA)	160
69	Example (Alarm – ProbLog2 [Fierens et al., 2015])	163
70	Example (Alarm – grounding – ProbLog2 [Fierens et al., 2015])	165
71	Example (Smokers – ProbLog [Fierens et al., 2015])	165
72	Example (Alarm – Boolean formula – ProbLog2 [Fierens et al., 2015])	167
73	Example (Symbolic derivation)	188
74	Example (Success functions of <i>msw</i> atoms)	190
75	Example (Join operation)	190
76	Example (Projection operation)	191
77	Example (Integration of a success function)	192
78	Example (Success function of a goal)	193
79	Example (Running example for lifted inference – ProbLog)	196
80	Example (Running example – PFL program)	197
81	Example (Translation of a ProbLog program into PFL) . . .	203
82	Example (ProbLog program to PFL – LP ²)	204
83	Example (ProbLog program to PFL – aggregation parfactors)	206
84	Example (ProbLog program to Skolem normal form)	209
85	Example (Path – ProbLog – iterative deepening)	213
86	Example (Path – ProbLog – <i>k</i> -best)	215
87	Example (Epidemic – LPAD)	219
88	Example (Machine diagnosis problem [Michels et al., 2016])	226
89	Example (Machine diagnosis problem – approximate inference – [Michels et al., 2016])	227
90	Example (Generative model)	236
91	Example (Gaussian mixture – sampling arguments – <code>cplint</code>)	238
92	Example (Possibilistic logic program)	240

93	Example (Remaining dry [Van den Broeck et al., 2010]) . . .	242
94	Example (Continuation of Example 93)	243
95	Example (Continuation of Example 93)	246
96	Example (Viral marketing [Van den Broeck et al., 2010]) . . .	249
97	Example (Alarm – aProbLog [Kimmig et al., 2011b])	253
98	Example (Bloodtype – PRISM [Sato et al., 2017])	259
99	Example (Epidemic – LPAD – EM)	271
100	Example (ILP problem)	284
101	Example (Examples of theta subsumption)	285
102	Example (Bottom clause example)	286

List of Definitions

1	Definition ($T_{\mathcal{P}}$ operator)	11
2	Definition ($OpFalse_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalse_{\mathcal{I}}^{\mathcal{P}}$ operators)	17
3	Definition (Iterated fixed point)	17
4	Definition (Acyclic, stratified and locally stratified programs)	20
5	Definition (Reduction)	21
6	Definition (Stable model)	21
7	Definition (Algebra)	23
8	Definition (σ -algebra)	23
9	Definition (Minimal σ -algebra)	23
10	Definition (Probability measure)	24
11	Definition (Finitely additive probability measure)	24
12	Definition (Random variable)	25
13	Definition (Cumulative distribution and probability density)	26
14	Definition (Product σ -algebra and product space)	30
15	Definition (Conditional probability)	30
16	Definition (Independence and conditional independence)	32
17	Definition (d-separation [Murphy, 2012])	35
18	Definition (Probability tree – positive case)	66
19	Definition (Hypothetical derivation sequence)	69
20	Definition (Probability tree – general case)	69
21	Definition (Valid CP-theory)	70
22	Definition (Parameterized two-valued interpretations)	102
23	Definition (Parameterized three-valued interpretations)	103
24	Definition ($OpTrue_{\mathcal{I}}^{\mathcal{P}}(Tr)$ and $OpFalse_{\mathcal{I}}^{\mathcal{P}}(Fa)$)	104
25	Definition (Iterated fixed point for probabilistic programs)	104
26	Definition (Infinite-dimensional product σ -algebra and space)	112
27	Definition (Valid program [Gutmann et al., 2011c])	120
28	Definition ($ST_{\mathcal{P}}$ operator [Gutmann et al., 2011c])	122
29	Definition ($ST_{\mathcal{P}}$ operator [Nitti et al., 2016])	122
30	Definition ($ST_{\mathcal{P}}$ operator – <code>cp lint</code>)	129

31	Definition (Probabilistic Constraint Logic Theory [Michels et al., 2015])	131
32	Definition (Credal set specification)	135
33	Definition (Conditional probability bounds)	138
34	Definition (Parameterized interpretation [Vlasselaer et al., 2015, 2016])	176
35	Definition (T_{CP} operator [Vlasselaer et al., 2015, 2016])	176
36	Definition (Fixpoint of T_{CP} [Vlasselaer et al., 2015, 2016])	177
37	Definition (Symbolic derivation [Islam et al., 2012b])	187
38	Definition (Derivation variables [Islam et al., 2012b])	189
39	Definition (Join operation)	190
40	Definition (Projection of a success function)	191
41	Definition (Integration of a success function)	191
42	Definition (Marginalization of a success function)	192
43	Definition (Success function of a goal)	193
44	Definition (Semiring)	250
45	Definition (aProbLog [Kimmig et al., 2011b])	251
46	Definition (PRISM parameter learning problem)	259
47	Definition (LLPAD Parameter learning problem)	266
48	Definition (Mutually exclusive bodies)	266
49	Definition (LeProbLog parameter learning problem)	267
50	Definition (EMBLEM parameter learning problem)	270
51	Definition (LFI-ProbLog learning problem)	280
52	Definition (Inductive Logic Programming – learning from entailment)	283
53	Definition (ALLPAD Structure learning problem)	287
54	Definition (Theory compression)	289
55	Definition (ProbFOIL/ProbFoil+ learning problem [Raedt et al., 2015])	290

List of Theorems

Theorem

1	Theorem (WFS for locally stratified programs [Van Gelder et al., 1991])	21
2	Theorem (WFS total model vs stable models)	21
3	Theorem (WFS vs stable models)	22
4	Theorem (Existence of a pairwise incompatible set of composite choices [Poole, 2000])	93
5	Theorem (Equivalence of the probability of two equivalent pairwise incompatible finite set of finite composite choices [Poole, 1993a])	94
6	Theorem (Algebra of a program)	95
7	Theorem (Finitely additive probability space of a program)	95
8	Theorem (Probability space of a program)	100
9	Theorem (Soundness and completeness of $IFPP^P$)	109
10	Theorem (Well-definedness of the distribution semantics)	109
11	Theorem (Equivalence with Sato and Kameya's definition)	113
12	Theorem (Conditions for exact inference of probability bounds [Michels et al., 2015])	140
13	Theorem (Model and weight equivalence [Fierens et al., 2015])	167
14	Theorem (Parameters as relative frequency)	266

Proposition

1	Proposition (Monotonic Mappings Have a Least and Greatest Fixpoint)	3
2	Proposition (Monotonicity of $OpTrueP_I^P$ and $OpFalseP_I^P$)	104
3	Proposition (Monotonicity of $IFPP^P$)	105
4	Proposition (Valid DC Program [Gutmann et al., 2011c])	121

5	Proposition (Conditions for exact inference [Michels et al., 2015])	134
6	Proposition (Computation of probability bounds)	136
7	Proposition (Conditional probability bounds formulas [Michels et al., 2015])	139

Lemma

1	Lemma (Infinite Product)	91
2	Lemma (σ -algebra of a Program)	99
3	Lemma (Existence of the limit of the measure of countable union of countable composite choices)	100
4	Lemma (Soundness of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$)	105
5	Lemma (Soundness of $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$)	106
6	Lemma (Partial evaluation)	106
7	Lemma (Model equivalence)	107
8	Lemma (Soundness of $IFPP^{\mathcal{P}}$)	107
9	Lemma (Completeness of $IFPP^{\mathcal{P}}$)	107
10	Lemma (Elements of $\Psi_{\mathcal{F}}$ as Countable Unions)	112
11	Lemma ($\Gamma_{\mathcal{F}}$ is Bijective)	113
12	Lemma (Correctness of the ProbLog Program Transformation)	166

List of Abbreviations

ADD	Algebraic Decision Diagram
AI	Artificial Intelligence
APLP	Annotated Probabilistic Logic Program
ASP	Answer Set Programming
BDD	Binary Decision Diagram
BLP	Bayesian Logic Program
BN	Bayesian Network
CBDD	Complete Binary Decision Diagram
CLP	Constraint Logic Programming
CNF	Conjunctive Normal Form
CPT	Conditional Probability Table
DC	Distributional Clauses
DCG	Definite Clause Grammar
d-DNNF	Deterministic Decomposable Negation Normal Form
DNF	Disjunctive Normal Form
DP	Dirichlet Process
DS	Distribution Semantics
EM	Expectation Maximization
ESS	Expected Sufficient Statistics
FED	Factored Explanation Diagram
FG	Factor Graph
GP	Gaussian Process
HMM	Hidden Markov Model
HPT	Hybrid Probability Tree
ICL	Independent Choice Logic
IHPMC	Iterative Hybrid Probabilistic Model Counting
IID	independent and identically distributed
ILP	Inductive Logic Programming
KBMC	Knowledge Base Model Construction
LDA	Latent Dirichlet Allocation
LL	Log Likelihood

LPAD	Logic Program with Annotated Disjunctions
MCMC	Markov Chain Monte Carlo
MDD	Multivalued Decision Diagram
MLN	Markov Logic Network
MN	Markov Network
NLP	Natural Language Processing
NNF	Negation Normal Form
PCFG	Probabilistic Context-Free Grammar
PCLP	Probabilistic Constraint Logic Programming
PFL	Prolog Factor Language
PHA	Probabilistic Horn Abduction
PHPT	Partially evaluated Hybrid Probability Tree
PILP	Probabilistic Inductive Logic Programming
PLCG	Probabilistic Left Corner Grammar
PLP	Probabilistic Logic Programming
POS	Part-of-Speech
PPDF	Product Probability Density Function
PPR	Personalized PageRank
PRV	Parameterized Random Variable
QSAR	Quantitative Structure–Activity Relationship
SDD	Sentential Decision Diagram
SLP	Stochastic Logic Program
WFF	Well-Formed Formula
WFM	Well-Founded Model
WFOMC	Weighted First-Order Model Counting
WFS	Well-Founded Semantics
WMC	Weighted Model Counting

Symbols

\mathbb{N}	natural numbers, i.e., non-negative integers $\{0, 1, 2, \dots\}$
\mathbb{N}_1	positive integers $\{1, 2, \dots\}$
\mathbb{R}	real numbers
$\mathbb{P}(S)$	powerset of set S
Ω	ordinal numbers
ω	first infinite ordinal
P	logic program

$\text{lhm}(P)$	least Herbrand model of P
\mathcal{U}	Herbrand universe
\mathcal{B}	Herbrand base
$\text{lfp}(T)$	least fixpoint of mapping T
$\text{gfp}(T)$	greatest fixpoint of mapping T
$\text{glb}(X)$	greatest lower bound of partially ordered set X
$\text{lub}(X)$	least upper bound of partially ordered set X
X, Y, \dots	logical variables
$\mathbf{X}, \mathbf{Y}, \dots$	vectors of logical variables
$\mathbf{x}, \mathbf{y}, \dots$	logical constants
ϕ, ψ, \dots	factors
a, b, \dots	logical atoms
μ	probability measure
X, Y, \dots	random variables
x, y, \dots	values assigned to random variables
$\mathbf{X}, \mathbf{Y}, \dots$	vectors of random variables
$\mathbf{x}, \mathbf{y}, \dots$	vectors of values of random variables
X, Y, \dots	parameterized random variable or parfactor
$\mathbf{X}, \mathbf{Y}, \dots$	vectors of parfactors
\mathcal{P}	probabilistic program
(f, θ, k)	atomic choice
κ	composite choice
σ	selection
w_σ	world identified by selection σ
w	world
$W_{\mathcal{P}}$	set of all worlds for program \mathcal{P}
K	set of composite choice
ω_κ	set of worlds compatible with composite choice κ
ω_K	set of worlds compatible with composite choice K
$\Omega_{\mathcal{P}}$	event space for program \mathcal{P}

1

Preliminaries

This chapter provides basic notions of logic programming and graphical models that are needed for the book. After a short introduction of a few mathematical concepts, the chapter presents logic programming and the various semantics for negation. Then it provides a brief recall of probability theory and graphical models.

For a more in-depth treatment of logic programming see [Lloyd, 1987; Sterling and Shapiro, 1994] and of graphical models see [Koller and Friedman, 2009].

1.1 Orders, Lattices, Ordinals

A *partial order* \leq is a reflexive, antisymmetric, and transitive relation. A *partially ordered set* S is a set with a partial order \leq . For example, the natural numbers \mathbb{N} (non-negative integers), the positive integers \mathbb{N}_1 , and the real numbers \mathbb{R} with the standard less-than-or-equal relation \leq are partially ordered sets and so is the powerset $\mathbb{P}(S)$ (the set of all subsets) of a set S with the inclusion relation between subsets \subseteq . $a \in S$ is an *upper bound* of a subset X of S if $x \leq a$ for all $x \in X$ and $b \in S$ is a *lower bound* of X if $b \leq x$ for all $x \in X$. If it also holds that $a \in X$ and $b \in X$, then a is called the *largest element* of X and b the *smallest element* of X .

An element $a \in S$ is the *least upper bound* of a subset X of S if a is an upper bound of X and, for all upper bounds a' of X , $a \leq a'$. An element $b \in S$ is the *greatest lower bound* of a subset X of S if b is a lower bound of X and, for all lower bounds b' of X , $b' \leq b$. The least upper bound of X may not exist. If it does, it is unique and we denote it with $\text{lub}(X)$. Similarly, the greatest lower bound of X may not exist. If it does, it is unique and is denoted by $\text{glb}(X)$. For example, for $\mathbb{P}(S)$ and $X \subseteq \mathbb{P}(S)$, $\text{lub}(X) = \bigcup_{x \in X} x$, and $\text{glb}(X) = \bigcap_{x \in X} x$.

2 Preliminaries

A partially ordered set L is a *complete lattice* if $\text{lub}(X)$ and $\text{glb}(X)$ exist for every subset X of L . We denote with \top the *top element* $\text{lub}(L)$ and with \perp the *bottom element* $\text{glb}(L)$ of the complete lattice L . For example, the powerset is a complete lattice.

A relation $<$ defined by $a < b$ iff $a < b$ and $a \neq b$ is associated with any partial order \leq on S .

A partial order \leq on a set S is a *total order* if it is *total*, i.e., for any $a, b \in S$, either $a \leq b$ or $b \leq a$. A set S with a total order is called a *totally ordered set*. A set S is *well-ordered* if it has a total order \leq such that every subset of S has a smallest element. The set \mathbb{N} is well-ordered by its usual order, the real line \mathbb{R} is not.

A function $f : A \rightarrow B$ is *one-to-one* if $f^{-1}(\{b\})$ is a set containing a single element. f is *onto* B if $f(A) = B$. A set A is *equipotent* with a set B iff there is a one-to-one function f from A onto B . Equipotency captures the intuitive notion of having the same number of elements.

A set S is *denumerable* iff S is equipotent with \mathbb{N} . A set S is *countable* iff it is finite or denumerable, S is *uncountable* otherwise.

Ordinal numbers are a generalization of natural numbers. The set Ω of *ordinal numbers* is well-ordered. We call its elements *ordinals* and we denote them by Greek smallcase letters. Since Ω is well-ordered, it has a smallest element, that we denote with 0. If $\alpha < \beta$, we say that α is a *predecessor* of β and β is a *successor* of α . α is the *immediate predecessor* of β if it is the largest ordinal smaller than β and β is the *immediate successor* of α if it is the largest ordinal smaller than α . Every ordinal α has an immediate successor, denoted with $\alpha + 1$. Some ordinals have predecessors but no immediate predecessor, which are called *limit ordinals*. The others are called *successor ordinals*. We denote the immediate successor of the least element 0 with 1, the immediate successor of 1 with 2, and so on. So the first elements 0, 1, 2, ... of Ω are the naturals. Since Ω is well-ordered, there is a smallest ordinal larger than 0, 1, 2, ... that we denote with ω . It is the *first infinite ordinal* and is countable, We can form its successors as $\omega + 1, \omega + 2, \dots$, effectively adding a “copy” of \mathbb{N} to the tail of ω . The smallest ordinal larger than $\omega + 1, \omega + 2, \dots$ is called 2ω and we can continue in this way building $3\omega, 4\omega$, and so on.

The smallest ordinal larger than these is ω^2 . We can repeat the process countably many times obtaining ω^3, ω^4 , and so on.

A canonical representation of the ordinals (the so-called *von Neumann ordinals*) sees each ordinal as the set of its predecessors, so $0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, \{\emptyset\}\}, 3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \dots$. In this case, the order is set membership.

The sequence of ordinals is also called *transfinite*. Mathematical induction is extended to the ordinals with the *principle of transfinite induction*. Suppose $P(\alpha)$ is a property defined for all ordinals $\alpha \in \Omega$. To prove that P is true for all ordinals by transfinite induction, we need to assume the fact that $P(\beta)$ is true for all $\beta < \alpha$ and prove that $P(\alpha)$ is true. Proofs by transfinite induction usually consider three cases: that α is 0, a successor ordinal, or a limit ordinal.

See [Srivastava, 2013] for a full formal definition of ordinal numbers and [Willard, 1970; Hitzler and Seda, 2016] for accessible introductions.

1.2 Mappings and Fixpoints

A function $T : L \rightarrow L$ from a lattice L to itself is a *mapping*. A mapping T is *monotonic* if $T(x) \leq T(y)$, for all x and y such that $x \leq y$. $a \in L$ is a *fixpoint* of T if $T(a) = a$. $a \in L$ is the *least fixpoint* of T if a is a fixpoint and, for all fixpoints b of T , it holds that $a \leq b$. Similarly, we define the *greatest fixpoint*.

Consider a complete lattice L and a monotonic mapping $T : L \rightarrow L$. We define the *increasing ordinal powers* of T as:

- $T \uparrow 0 = \perp$;
- $T \uparrow \alpha = T(T \uparrow (\alpha - 1))$, if α is a successor ordinal;
- $T \uparrow \alpha = \text{lub}(\{T \uparrow \beta \mid \beta < \alpha\})$, if α is a limit ordinal;

and the *decreasing ordinal powers* of T as:

- $T \downarrow 0 = \top$;
- $T \downarrow \alpha = T(T \downarrow (\alpha - 1))$, if α is a successor ordinal;
- $T \downarrow \alpha = \text{glb}(\{T \downarrow \beta \mid \beta < \alpha\})$, if α is a limit ordinal.

The Knaster–Tarski theorem [Knaster and Tarski, 1928; Tarski, 1955] states that if L is complete lattice and T a monotonic mapping, then the set of fixpoints of T in L is also a lattice. An important consequence of the theorem is the following proposition.

Proposition 1 (Monotonic Mappings Have a Least and Greatest Fixpoint). *Let L be a complete lattice and $T : L \rightarrow L$ be monotonic. Then T has a least fixpoint, $\text{lfp}(T)$ and a greatest fixpoint $\text{gfp}(T)$.*

A sequence $\{x_\alpha \mid \alpha \in \Omega\}$ is *increasing* if $x_\beta \leq x_\alpha$ for all $\beta \leq \alpha$ and is *decreasing* if $x_\alpha \leq x_\beta$ for all $\beta \leq \alpha$.

The increasing and decreasing ordinal powers of a monotonic mapping form an increasing and a decreasing sequence, respectively. Let us prove it

4 Preliminaries

for $T \uparrow \alpha$ by transfinite induction. If α is a successor ordinal, $T \uparrow (\alpha - 2) \leq T \uparrow (\alpha - 1)$ for the inductive hypothesis. By the monotonicity of T , $T(T \uparrow (\alpha - 2)) \leq T(T \uparrow (\alpha - 1))$, so $T \uparrow (\alpha - 1) \leq T \uparrow \alpha$. Since $T \uparrow \beta \leq T \uparrow (\alpha - 1)$ for all $\beta \leq \alpha - 1$, for the transitivity of \leq , the thesis is proved. If α is a limit ordinal, $T \uparrow \alpha = \text{lub}(\{T \uparrow \beta \mid \beta < \alpha\})$, so the thesis is proved. It can be proved for $T \downarrow \alpha$ similarly. Note that in general the fact that T is a monotonic mapping does not imply that $x \leq T(x)$ for all x .

1.3 Logic Programming

This section provides some basic notions of first-order logic languages and logic programming.

A *first-order logic language* is defined by an alphabet that consists of the following sets of symbols: variables, constants, functions symbols, predicate symbols, logical connectives, quantifiers, and punctuation symbols. The last three are the same for all logic languages. The connectives are \neg (negation), \wedge (conjunction), \vee (disjunction), \leftarrow (implication), and \leftrightarrow (equivalence); the quantifiers are the existential quantifier \exists and the universal quantifier \forall , and the punctuation symbols are “(”, “)”, and “,”.

Well-Formed Formulas (WFFs) of the language are the syntactically correct clauses of the language and are inductively defined by combining elementary formulas, called *atomic formulas*, by means of logical connectives and quantifiers. On their turn, atomic formulas are obtained by applying the predicates symbols to elementary terms.

A *term* is defined recursively as follows: a variable is a term, a constant is a term, if f is a function symbol with *arity* n and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term. An *atomic formula* or *atom* a is the application of a predicate symbol p with *arity* n to n terms: $p(t_1, \dots, t_n)$.

The following notation for the symbols will be adopted: predicates, functions, and constants start with a lowercase letter, while variables start with an uppercase letter (as in the Prolog programming language, see below). So x, y, \dots are constants and X, Y, \dots are variables. Bold typeface is used throughout the book for vectors, so $\mathbf{X}, \mathbf{Y}, \dots$ are vectors of logical variables.

An example of a term is *mary*, a constant, or *father(mary)*, a complex term, where *father* is a function symbol with *arity* 1. An example of an atom is *parent(father(mary), mary)* where *parent* is a predicate with *arity* 2. To take into account the *arity*, function symbols and predicates are usually indicated as *father/1* and *parent/2*. In this case, the symbols *father*

and *parent* are called functors. Atoms are indicated with lowercase letters a, b, \dots

A WFF is defined recursively as follows:

- every atom a is a WFF;
- if A and B are WFFs, then also $\neg A$, $A \wedge B$, $A \vee B$, $A \leftarrow B$, $A \leftrightarrow B$ are WFFs (possibly enclosed in balanced brackets);
- if A is WFF and X is a variable, $\forall X A$ and $\exists X A$ are WFF.

A *variant* ϕ' of a formula ϕ is obtained by renaming all the variables in ϕ .

The class of formulas called clauses has important properties. A *clause* is a formula of the form

$$\forall X_1 \forall X_2 \dots \forall X_s (a_1 \vee \dots \vee a_n \vee \neg b_1 \vee \dots \vee \neg b_m)$$

where each a_i, b_i are atoms and X_1, X_2, \dots, X_s are all the variables occurring in $(a_1 \vee \dots \vee a_n \vee \neg b_1 \vee \dots \vee \neg b_m)$. The clause above can also be represented as follows:

$$a_1 ; \dots ; a_n \leftarrow b_1, \dots, b_m$$

where commas stand for conjunctions and semicolons for disjunctions. The part preceding the symbol \leftarrow is called the *head* of the clause, while the part following it is called the *body*. An atom or the negation of an atom is called a *literal*. A *positive literal* is an atom, and a *negative literal* is the negation of an atom. Sometimes, clauses will be represented by means of a set of literals:

$$\{a_1, \dots, a_n, \neg b_1, \dots, \neg b_m\}$$

An example of a clause is

$$male(X) ; female(X) \leftarrow human(X).$$

A clause is a *denial* if it has no positive literal, *definite* if it has one positive literal, and *disjunctive* if it has more than one positive literal. A *Horn clause* is either a definite clause or a denial. A *fact* is a definite clause without negative literals, the \leftarrow symbol is omitted for facts. A clause C is *range-restricted* if and only if the variables appearing in the head are a subset of those in the body. A *definite logic program* P is a finite set of definite clauses.

Examples of a definite clause, a denial, and a fact are, respectively:

$$\begin{aligned} & human(X) \leftarrow female(X) \\ & \leftarrow male(X), female(X) \\ & female(mary) \end{aligned}$$

6 Preliminaries

In this book, I will also present clauses in monospaced font especially when they form programs that can be directly input into a logic programming system (*concrete syntax*). In that case, the implication symbol is represented by `:-` as in

```
human(X) :- female(X).
```

In logic programming, another type of negation is also taken into account, namely, the so-called *default negation* \sim . The formula $\sim a$ where a is an atom is called a *default negative literal*, sometimes abbreviated as simply *negative literal* if the meaning is clear from the context. A *default literal* is either an atom (a *positive literal*) or a default negative literal. Again the word default can be dropped if the meaning is clear from the context.

A *normal clause* is a clause of the form

$$a \leftarrow b_1, \dots, b_m$$

where each b_i is a default literal. A *normal logic program* is a finite set of normal clauses. Default negation in the concrete syntax is represented either with `\+` (Prolog, see page 9) or `not` (Answer Set Programming, see Section 1.4.3). In this book, we will use the Prolog convention and use `\+`.

A *substitution* $\theta = \{X_1/t_1, \dots, X_k/t_k\}$ is a function mapping variables to terms. $\theta = \{X/father(mary), Y/mary\}$ is an example of a substitution. The *application* θ of a substitution θ to a formula ϕ means replacing all the occurrences of each variable X_j in ϕ by the same term t_j . So $parent(X, Y)\theta$ is, for example, $parent(father(mary), mary)$.

An *antisubstitution* $\theta^{-1} = \{t_1/V_1, \dots, t_m/V_m\}$ is a function mapping terms to variables. In an antisubstitution, the terms must be such that each of them must not be equal or a sub-term of another one; otherwise, the substitution process is not well-defined. An antisubstitution is the inverse of substitution $\theta = \{V_1/t_1, \dots, V_m/t_m\}$ if the terms satisfy the above constraint. In this case, for any formula ϕ , $\phi\theta\theta^{-1} = \phi$.

A *ground clause* (term) is a clause (term) without variables. A substitution θ is *grounding* for a formula ϕ if $\phi\theta$ is ground.

The *Herbrand universe* \mathcal{U} of a language or a program is the set of all the ground terms that can be obtained by combining the symbols in the language or program. The *Herbrand base* \mathcal{B} of a language or a program is the set of all possible ground atoms built with the symbols in the language or program. Sometimes they will be indicated with \mathcal{U}_P and \mathcal{B}_P where P is the program. The *grounding ground*(P) of a program P is obtained by replacing the variables of clauses in P with terms from \mathcal{U}_P in all possible ways.

If P does not contain function symbols, then \mathcal{U}_P is equal to the set of constants and is finite; otherwise, it is infinite (e.g., if P contains constant 0 and function symbol $s/1$, then $\mathcal{U}_P = \{0, s(0), s(s(0)), \dots\}$). Therefore, if P does not contain function symbols, $\text{ground}(P)$ is finite and is infinite if P contains function symbols and at least one variable. The language of programs without function symbols is called *Datalog*.

The semantics of a set of formulas can be defined in terms of interpretations and models. We will here consider the special case of *Herbrand interpretations* and *Herbrand models* that are sufficient for giving a semantics to sets of clauses. For a definition of interpretations and models in the general case, see [Lloyd, 1987]. A *Herbrand interpretation* or *two-valued interpretation* I is a subset of the Herbrand base, i.e., $I \subseteq \mathcal{B}$. Given a Herbrand interpretation, it is possible to assign a truth value to formulas according to the following rules. A ground atom $p(t_1, t_2, \dots, t_n)$ is true under the interpretation I if and only if $p(t_1, t_2, \dots, t_n) \in I$. A conjunction of atomic formulas b_1, \dots, b_m is true in I if and only if $b_1, \dots, b_m \subseteq I$. A ground clause $a_1 ; \dots ; a_n \leftarrow b_1, \dots, b_m$ is true in an interpretation I if and only if at least one of the atoms of the head is true in the case in which the body is true. A clause C is true in an interpretation I if and only if all of its ground instances with terms from \mathcal{U} are true in I . A set of clauses Σ is true in an interpretation I if and only if all the clauses $C \in \Sigma$ are true.

A two-valued interpretation I represents the set of true atoms, so a is true in I if $a \in I$ and is false if $a \notin I$. The set Int2 of two-valued interpretations for a program P forms a complete lattice where the partial order \leq is given by the subset relation \subseteq . The least upper bound and greatest lower bound are thus $\text{lub}(X) = \bigcup_{I \in X} I$ and $\text{glb}(X) = \bigcap_{I \in X} I$. The bottom and top element are, respectively, \emptyset and \mathcal{B}_P .

An interpretation I *satisfies* a set of clauses Σ , notation $I \models \Sigma$, if Σ is true in I ; we also say that I is a *model* of Σ . A set of clauses is *satisfiable* if it is satisfied by some interpretation, *unsatisfiable* otherwise. If all models of a set of clauses Σ are also models of a clause C , we say that Σ *logically entails* C or C is a *logical consequence* of Σ , and we write $\Sigma \models C$. We use the same symbol for the entailment relation and for the satisfaction relation between interpretations and formulas in order to follow the standard logic practice. In cases where this may cause misunderstanding, the intended meaning will be indicated in words.

Herbrand interpretations and models are sufficient for giving a semantics to sets of clauses in the following sense: a set of clauses is unsatisfiable if and only if it does not have a Herbrand model, a consequence of Herbrand's

theorem [Herbrand, 1930]. For sets of definite clauses, Herbrand models are particularly important because they have the relevant property that the intersection of a set of Herbrand models for a set of definite clauses P is still a Herbrand model of P . The intersection of all the Herbrand models of P is called the *minimal Herbrand model* of P and is represented with $\text{lh}(P)$. The least Herbrand model of P always exists and is unique. The *model-theoretic semantics* of a program P is the set of all ground atoms that are logical consequences of P . The least Herbrand model provides the model theoretic semantics for P : $P \models a$ if and only if $a \in \text{lh}(P)$ where a is a ground atom.

For example, the program P below

$$\begin{aligned} \text{human}(X) &\leftarrow \text{female}(X) \\ \text{female}(\text{mary}) \end{aligned}$$

has the $\text{lh}(P) = \{\text{female}(\text{mary}), \text{human}(\text{mary})\}$.

A *proof procedure* is an algorithm for checking whether a formula is provable from a theory. If formula ϕ is provable from the set of formulas Σ , we write $\Sigma \vdash \phi$. Two important properties of proof procedures are *soundness* and *completeness*. A proof procedure is *sound*, with respect to the model-theoretic semantics, if $\Sigma \models \phi$ whenever $\Sigma \vdash \phi$; it is *complete* if $\Sigma \vdash \phi$ whenever $\Sigma \models \phi$.

A proof procedure for clausal logic that is particularly suitable to be automated on a computer is *resolution* [Robinson, 1965]. The resolution inference rule allows one to prove, from two clauses $F_1 \vee l_1$ and $F_2 \vee \neg l_2$ where F_1 and F_2 are disjunctions of literals, the clause $(F_1 \vee F_2)\theta$, where θ is the *most general unifier* of l_1 and l_2 , i.e., the minimal substitution such that $l_1\theta = l_2\theta$. For definite clauses, this consists in matching the head of one clause with a literal in the body of another. To prove a conjunction of literals ϕ (a *query* or *goal*) from a set of clauses Σ , ϕ is negated obtaining $\leftarrow \phi$ and added to Σ : if the empty denial \leftarrow can be obtained from $\Sigma \cup \{\leftarrow \phi\}$ using a sequence of applications of the resolution inference rule, then ϕ is provable from Σ , $\Sigma \vdash \phi$. For example, if we want to prove that $P \vdash \text{human}(\text{mary})$, we need to prove that \leftarrow can be derived from $P \cup \{\leftarrow \text{human}(\text{mary})\}$. In this case, from $\text{human}(X) \vee \neg \text{female}(X)$ and $\text{female}(\text{mary})$, we can derive $\text{human}(\text{mary})$ by resolution.

Logic programming was originally proposed by considering Horn clauses and adopting a particular version of resolution, *SLD resolution*, [Kowalski, 1974]. SLD resolution builds a sequence of formulas $\phi_1, \phi_2, \dots, \phi_n$, where $\phi_1 = \leftarrow \phi$, that is called a *derivation*. At each step, the new formula ϕ_{i+1} is obtained by resolving the previous formula ϕ_i with a variant of a clause from

the program P . If no resolution can be performed, then $\phi_{i+1} = \text{fail}$ and the derivation cannot be further extended. If $\phi_n = \leftarrow$, the derivation is *successful*; if $\phi_n = \text{fail}$, the derivation is *unsuccessful*. The query ϕ is proved (*succeeds*) if there exists a successful derivation for it and *fails* otherwise. SLD resolution was proven to be sound and complete (under certain conditions) for Horn clauses (the proofs can be found in [Lloyd, 1987]).

A particular Logic programming language is defined by choosing a rule for the selection of the literal in the current formula to be reduced at each step (*selection rule*) and by choosing a search strategy that can be either depth first or breadth first. In the *Prolog* [Colmerauer et al., 1973] programming language, the selection rule selects the *left-most* literal in the current goal and the search strategy is depth first with chronological backtracking. Prolog builds an *SLD tree* to answer queries: nodes are formulas and each path from the root to a leaf is an SLD derivation; branching occurs when there is more than one clause that can be resolved with the goal of a node. Prolog adopts an extension of SLD resolution called *SLDNF resolution* that is able to deal with normal clauses by *negation as failure* [Clark, 1978]: a negative selected literal $\sim a$ is removed from the current goal if a proof for a fails. An *SLDNF tree* is an SLD tree where the literal $\sim a$ in a node is handled by building a nested tree for a : if the nested tree has no successful derivations, the literal $\sim a$ is removed from the node and derivation proceeds; otherwise, the node has fail as the only child.

If the goal ϕ contains variables, a *solution* to ϕ is a substitution θ obtained by composing the substitutions along a branch of the SLDNF tree that is successful. The *success set* of ϕ is the set of solutions of ϕ . If a normal program is range restricted, every successful SLDNF derivation for goal ϕ completely grounds ϕ [Muggleton, 2000a].

Example 1 (Path – Prolog). *The following program computes paths in a graph:*

```

path(X, X).
path(X, Y) ← edge(X, Z), path(Z, Y).
edge(a, b).
edge(b, c).
edge(a, c).

```

path(X, Y) is true if there is a path from X to Y in the graph where the edges are represented by facts for the predicate edge/2. This program computes the transitive closure of the relation edge. This is possible because the program contains an inductive definition, that of path/2.

Computing transitive closures is an example of a problem for which first order logic is not sufficient and a Turing-complete language is required. Inductive definitions provide Prolog this expressive power.

The first clause states that there is a path from a node to itself. The second states that there is a path from a node X to a node Y if there exists a node Z such that there is an edge from X to Z and there is a path from Z to Y . Variables appearing in the body only, such as Z above, become existentially quantified when the universal quantifier for them is moved to have the body as scope.

Figure 1.1 shows the SLD tree for the query $\text{path}(a, c)$. The labels of the edges indicate the most general unifiers used. The query has two successful derivations, corresponding to the paths from the root to the \leftarrow leaves.

Suppose we add the following clauses to the program

$$\begin{aligned} \text{ends}(X, Y) &\leftarrow \text{path}(X, Y), \sim \text{source}(Y). \\ \text{source}(X) &\leftarrow \text{edge}(X, Y). \end{aligned}$$

$\text{ends}(X, Y)$ is true if there is a path from X to Y and Y is a terminal node, i.e., it has no outgoing edges.

The SLDNF tree for the query $\text{ends}(b, c)$ is shown in Figure 1.2: to prove $\sim \text{source}(c)$, an SLDNF tree is built for $\text{source}(c)$, shown in the rectangle. Since the derivation of $\text{source}(c)$ fails, then $\sim \text{source}(c)$ succeeds and can be removed from the goal.

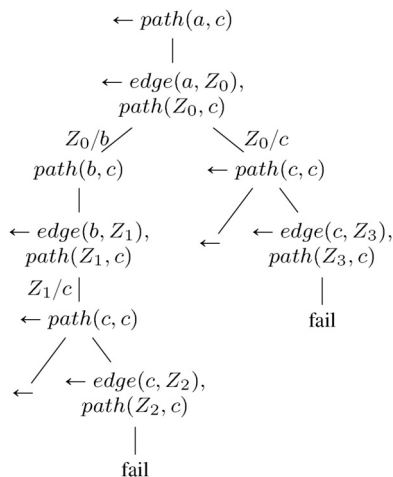


Figure 1.1 SLD tree for the query $\text{path}(a, c)$ from the program of Example 1.

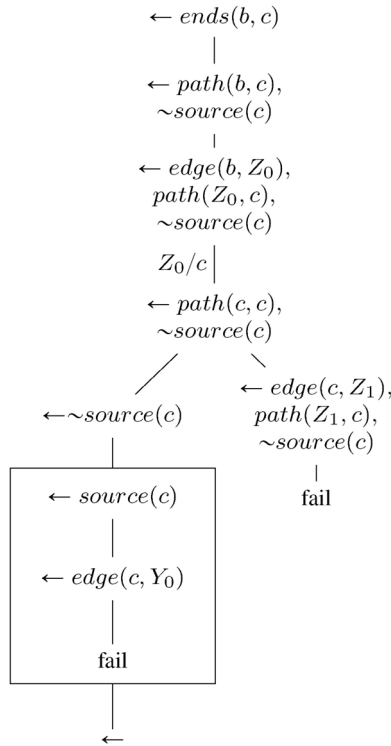


Figure 1.2 SLDNF tree for the query $ends(b, c)$ from the program of Example 1.

A ground atom is *relevant* with respect to a query ϕ if it occurs in some proof of an atom from $atoms(\phi)$, where $atoms(\phi)$ returns the set of atoms appearing in the conjunction of literals ϕ . A ground rule is relevant if it contains only relevant atoms. In Example 1, $path(a, b)$, $path(b, c)$, and $path(c, c)$ are relevant for $path(a, c)$, while $path(b, a)$, $path(c, a)$, and $path(c, b)$ aren't. For the query $ends(b, c)$, instead $path(b, c)$, $path(c, c)$, and $source(c)$ are relevant, while $source(b)$ is not.

Proof procedures provide a method for answering queries in a top-down way. We can also proceed bottom-up by computing all possible consequences of the program using the T_P or *immediate consequence operator*.

Definition 1 (T_P operator). For a definite program P , we define the operator $T_P : Int2 \rightarrow Int2$ as

$$T_P(I) = \{a \mid \text{there is a clause } b \leftarrow l_1, \dots, l_n \text{ in } P, \text{ a grounding substitution } \theta \text{ such that } a = b\theta \text{ and, for every } 1 \leq i \leq n, l_i\theta \in I\}.$$

The T_P operator is such that its least fixpoint is equal to the least Herbrand model of the program: $\text{lfp}(T_P) = \text{lhm}(P)$.

It can be shown that the least fixpoint of the T_P operator is reached at most at the first infinite ordinal ω [Hitzler and Seda, 2016].

In logic programming, data and code take the same form, so it is easy to write programs that manipulate code. For example, *meta-interpreters* or programs for interpreting other programs are particularly simple to write. A meta-interpreter for pure Prolog is [Sterling and Shapiro, 1994]:

```
solve(true) .
solve(A, B) :-
    solve(A) ,
    solve(B) .
solve(Goal) :-
    clause(Goal, Body) ,
    solve(Body) .
```

where $\backslash=$ is a predicate that succeeds if the two arguments do not unify, $\text{clause}(\text{Goal}, \text{Body})$ is a predicate that succeeds if $\text{Goal} \text{ :- } \text{Body}$ is a clause of the program, and each occurrence of $_$ indicates a distinct *anonymous variable*, i.e., a variable that is there as a placeholder and for which we don't care about its value.

Pure Prolog is extended to include primitives for controlling the search process. An example is the *cut* which is a predicate $! / 0$ that always succeeds and cuts choice points. If we have the program

```
p(S1) :- A1 .
...
p(Sk) :- B, !, C .
...
p(Sn) :- An .
```

when the k -th clause is evaluated, if B succeeds, then $!$ succeeds and the evaluation proceeds with C . In the case of backtracking, all the alternative for B are eliminated, as well as all the alternatives for p provided by the clauses from the k -th to the n -th.

The cut is used in the implementation of an if-then-else construct in Prolog that is represented as

```
(B->C1;C2) .
```

The construct evaluates B; if it succeeds, it evaluates C1; if it fails, it evaluates C2.

The construct is implemented as

```
if_then_else(B, C1, C2) :- B, !, C1.
if_then_else(B, C1, C2) :- C2.
```

Prolog systems usually include many built-in predicates. The predicate `is/2` (usually written in infix notation) evaluates the second argument as an arithmetic expression and unifies the first argument with the result. So for example, `A is 3+2` is a goal that, when executed, unifies A with 5.

1.4 Semantics for Normal Logic Programs

In a normal program, the clauses can contain default negative literals in the body, so a general rule has the form

$$C = h \leftarrow b_1, \dots, b_n, \sim c_1, \dots, \sim c_m \quad (1.1)$$

where $h, b_1, \dots, b_n, c_1, \dots, c_m$ are atoms.

Many semantics have been proposed for normal logic programs; see [Apt and Bol, 1994] for a survey. Of them, the one based on Clark's completion, the Well-Founded Semantics (WFS), and the stable model semantics are the most widely used.

1.4.1 Program Completion

Program completion or *Clark's completion* [Clark, 1978] assigns a semantics to a normal program by building a first-order logical theory representing the meaning of the program. The idea of the approach is to formally model the fact that atoms not inferable from the rules in the program are to be regarded as false.

The *completion of a clause*

$$p(t_1, \dots, t_n) \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_l.$$

with variables Y_1, \dots, Y_d is the clause

$$p(X_1, \dots, X_n) \leftarrow \exists Y_1, \dots, \exists Y_d ((X_1 = t_1) \wedge \dots \wedge (X_n = t_n) \wedge b_1 \wedge \dots \wedge b_m \wedge \neg c_1 \wedge \dots \wedge \neg c_l)$$

14 Preliminaries

where $=$ is a new predicate symbol representing equality. If the program contains $k \geq 1$ clauses for predicate p , we obtain k formulas of the form

$$\begin{aligned} p(X_1, \dots, X_n) &\leftarrow E_1 \\ &\vdots \\ p(X_1, \dots, X_n) &\leftarrow E_k. \end{aligned}$$

The *completed definition* of p is then

$$\forall X_1, \dots, \forall X_n (p(X_1, \dots, X_n) \leftrightarrow E_1 \vee \dots \vee E_k).$$

If a predicate appearing in the program does not appear in the head of any clause, we add

$$\forall X_1, \dots, \forall X_n \neg p(X_1, \dots, X_n).$$

The predicate $=$ is constrained by the following formulas that form an *equality theory*:

1. $c \neq d$ for all pairs c, d of distinct constants,
2. $\forall (f(X_1, \dots, X_n) \neq g(Y_1, \dots, Y_m))$ for all pairs f, g of distinct function symbols,
3. $\forall (t[X] \neq X)$ for each term $t[X]$ containing X and different from X ,
4. $\forall ((X_1 \neq Y_1) \vee \dots \vee (X_n \neq Y_n) \rightarrow f(X_1, \dots, X_n) \neq f(Y_1, \dots, Y_n))$ for each function symbol f ,
5. $\forall (X = X)$,
6. $\forall ((X_1 = Y_1) \wedge \dots \wedge (X_n = Y_n) \rightarrow f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n))$ for each function symbol f ,
7. $\forall ((X_1 = Y_1) \wedge \dots \wedge (X_n = Y_n) \rightarrow p(X_1, \dots, X_n) \rightarrow p(Y_1, \dots, Y_n))$ for each predicate symbol p (including $=$).

Given a normal program P , the completion $comp(P)$ of P is the theory formed by the completed definition of each predicate of P and the equality theory.

Note that the completion of a program may be inconsistent. In this case, the theory has no models and everything is a logical consequence. This is a situation to be avoided so restrictions are imposed on the form of the program.

The idea of the semantics based on Clark's completion is to consider a conjunction of ground literals

$$b_1, \dots, b_m, \sim c_1, \dots, \sim c_l$$

true if

$$b_1 \wedge \dots \wedge b_m \wedge \neg c_1 \wedge \dots \wedge \neg c_l$$

is a logical consequence of $\text{comp}(P)$, i.e., if

$$\text{comp}(P) \models b_1 \wedge \dots \wedge b_m \wedge \neg c_1 \wedge \dots \wedge \neg c_l.$$

SLDNF resolution was proven sound and complete (under certain conditions) with respect to the Clark's completion semantics [Clark, 1978].

Example 2 (Clark's completion).

Consider the program P_1

$$b \leftarrow \sim a.$$

$$c \leftarrow \sim b.$$

$$c \leftarrow a.$$

Its completion $\text{comp}(P_1)$ is

$$b \leftrightarrow \neg a.$$

$$c \leftrightarrow \neg b \vee a.$$

$$\neg a$$

We see that $\text{comp}(P_1) \models \neg a, b, \neg c$ so $\sim a, b$ and $\sim c$ can be derived with SLDNF resolution. The SLDNF tree for query c is shown in Figure 1.3 and, as expected, returns false.

Consider the program P_2

$$p \leftarrow p.$$

Its completion $\text{comp}(P_2)$ is

$$p \leftrightarrow p.$$

Here $\text{comp}(P) \not\models p$ and $\text{comp}(P) \not\models \neg p$. SLDNF resolution for query p would loop forever. This shows that SLNDF does not handle well loops, in this case positive loops.

Consider the program P_3

$$p \leftarrow \sim p.$$

Its completion $\text{comp}(P_3)$ is

$$p \leftrightarrow \neg p.$$

which is inconsistent, so everything is a consequence. SLDNF resolution for query p would loop forever. SLNDF resolution does not handle well also loops through negation or negative loops.

1.4.2 Well-Founded Semantics

The Well-Founded Semantics (WFS) [Van Gelder et al., 1991] assigns a three-valued model to a program, i.e., it identifies a consistent three-valued interpretation as the meaning of the program.

A three-valued interpretation \mathcal{I} is a pair $\langle I_T, I_F \rangle$ where I_T and I_F are subsets of \mathcal{B}_P and represent, respectively, the set of true and false atoms.

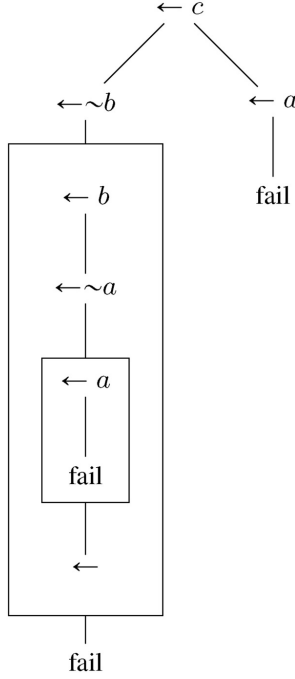


Figure 1.3 SLDNF tree for the query c from the program of Example 2.

So a is true in \mathcal{I} if $a \in I_T$ and is false in \mathcal{I} if $a \in I_F$, and $\sim a$ is true in \mathcal{I} if $a \in I_F$ and is false in \mathcal{I} if $a \in I_T$. If $a \notin I_T$ and $a \notin I_F$, then a assumes the third truth value, *undefined*. We also write $\mathcal{I} \models a$ if $a \in I_T$ and $\mathcal{I} \models \sim a$ if $a \in I_F$. A *consistent* three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ is such that $I_T \cap I_F = \emptyset$. The union of two three-valued interpretations $\langle I_T, I_F \rangle$ and $\langle J_T, J_F \rangle$ is defined as $\langle I_T, I_F \rangle \cup \langle J_T, J_F \rangle = \langle I_T \cup J_T, I_F \cup J_F \rangle$. The intersection of two three-valued interpretations $\langle I_T, I_F \rangle$ and $\langle J_T, J_F \rangle$ is defined as $\langle I_T, I_F \rangle \cap \langle J_T, J_F \rangle = \langle I_T \cap J_T, I_F \cap J_F \rangle$. Sometimes we represent a three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ as a single set of literals, i.e.,

$$\mathcal{I} = I_T \cup \{\sim a \mid a \in I_F\}.$$

The set $Int3$ of three-valued interpretations for a program P forms a complete lattice where the partial order \leq is defined as $\langle I_T, I_F \rangle \leq \langle J_T, J_F \rangle$ if $I_T \subseteq J_T$ and $I_F \subseteq J_F$. The least upper bound and greatest lower bound are defined as $\text{lub}(X) = \bigcup_{\mathcal{I} \in X} \mathcal{I}$ and $\text{glb}(X) = \bigcap_{\mathcal{I} \in X} \mathcal{I}$. The bottom and top element are, respectively, $\langle \emptyset, \emptyset \rangle$ and $\langle \mathcal{B}_P, \mathcal{B}_P \rangle$.

Given a three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$, we define the functions $true(\mathcal{I}) = I_T$, $false(\mathcal{I}) = I_F$, and $undef(\mathcal{I}) = \mathcal{B}_P \setminus I_T \setminus I_F$ that return the set of true, false and undefined atoms, respectively.

The WFS was given in [Van Gelder et al., 1991] in terms of the least fixpoint of an operator that is composed by two sub-operators, one computing consequences, and the other computing unfounded sets. We give here the alternative definition of the WFS of [Przymusinski, 1989] that is based on an iterated fixpoint.

Definition 2 ($OpFalse_{\mathcal{I}}^P$ and $OpTrue_{\mathcal{I}}^P$ operators). *For a normal program P , sets Tr and Fa of ground atoms, and a three-valued interpretation \mathcal{I} , we define the operators $OpTrue_{\mathcal{I}}^P : Int2 \rightarrow Int2$ and $OpFalse_{\mathcal{I}}^P : Int2 \rightarrow Int2$ as*

$$OpTrue_{\mathcal{I}}^P(Tr) = \{a \mid a \text{ is not true in } \mathcal{I}; \text{ and there is a clause } b \leftarrow l_1, \dots, l_n \text{ in } P, \text{ a grounding substitution } \theta \text{ such that } a = b\theta \text{ and for every } 1 \leq i \leq n \text{ either } l_i\theta \text{ is true in } \mathcal{I}, \text{ or } l_i\theta \in Tr\};$$

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

In words, the operator $OpTrue_{\mathcal{I}}^P(Tr)$ extends the interpretation \mathcal{I} to add the new true atoms that can be derived from P knowing \mathcal{I} and true atoms Tr , while $OpFalse_{\mathcal{I}}^P(Fa)$ computes new false atoms in P by knowing \mathcal{I} and false atoms Fa . $OpTrue_{\mathcal{I}}^P$ and $OpFalse_{\mathcal{I}}^P$ are both monotonic [Przymusinski, 1989], so they both have least and greatest fixpoints. An iterated fixpoint operator builds up *dynamic strata* by constructing successive three-valued interpretations as follows.

Definition 3 (Iterated fixed point). *For a normal program P , let $IFP^P : Int3 \rightarrow Int3$ be defined as*

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

IFP^P is monotonic [Przymusinski, 1989] and thus has a least fixpoint $\text{lfp}(IFP^P)$. The Well-Founded Model (WFM) $WFM(P)$ of P is $\text{lfp}(IFP^P)$. Let δ be the smallest ordinal such that $WFM(P) = IFP^P \uparrow \delta$. We refer to δ as the *depth* of P . The *stratum* of atom a is the least ordinal β such that $a \in IFP^P \uparrow \beta$ (where a may be either in the true or false component of $IFP^P \uparrow \beta$). Undefined atoms of the WFM do not belong to any stratum – i.e., they are not added to $IFP^P \uparrow \delta$ for any ordinal δ .

If $undef(WFM(P)) = \emptyset$, then the WFM is called *total* or *two-valued* and the program *dynamically stratified*.

Example 3 (WFS computation). *Let us consider program P_1 of Example 2:*

$$\begin{aligned} b &\leftarrow \sim a. \\ c &\leftarrow \sim b. \\ c &\leftarrow a. \end{aligned}$$

Its iterated fixpoint is

$$\begin{aligned} IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\ IFP^P \uparrow 1 &= \langle \emptyset, \{a\} \rangle; \\ IFP^P \uparrow 2 &= \langle \{b\}, \{a\} \rangle; \\ IFP^P \uparrow 3 &= \langle \{b\}, \{a, c\} \rangle; \\ IFP^P \uparrow 4 &= IFP^P \uparrow 3 = WFM(P_1). \end{aligned}$$

Thus, the depth of P_1 is 3 and the WFM of P_1 is given by

$$\begin{aligned} true(WFM(P_1)) &= \{b\} \\ undef(WFM(P_1)) &= \emptyset \\ false(WFM(P_1)) &= \{a, c\}. \end{aligned}$$

So $WFM(P_1)$ is two-valued and P_1 is dynamically stratified.

Let us consider program P_4 from [Przymusiński, 1989]

$$\begin{aligned} b &\leftarrow \sim a. \\ c &\leftarrow \sim b. \\ c &\leftarrow a, \sim p. \\ p &\leftarrow \sim q. \\ q &\leftarrow \sim p, b. \end{aligned}$$

Its iterated fixpoint is

$$\begin{aligned} IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\ IFP^P \uparrow 1 &= \langle \emptyset, \{a\} \rangle; \\ IFP^P \uparrow 2 &= \langle \{b\}, \{a\} \rangle; \\ IFP^P \uparrow 3 &= \langle \{b\}, \{a, c\} \rangle; \\ IFP^P \uparrow 4 &= IFP^P \uparrow 3 = WFM(P_4). \end{aligned}$$

So the depth of P_4 is 3 and the WFM of P_4 is given by

$$\begin{aligned} true(WFM(P_4)) &= \{b\} \\ undef(WFM(P_4)) &= \{p, q\} \\ false(WFM(P_4)) &= \{a, c\}. \end{aligned}$$

Consider now the program P_2 of Example 2:

$$p \leftarrow p.$$

Its iterated fixpoint is

$$\begin{aligned} IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\ IFP^P \uparrow 1 &= \langle \emptyset, \{p\} \rangle; \\ IFP^P \uparrow 2 &= IFP^P \uparrow 1 = WFM(P_2). \end{aligned}$$

P_2 is dynamically stratified and assigns value false to p . So positive loops are resolved by assigning value false to the atom.

Let us consider the program P_3 of Example 2:

$$p \leftarrow \sim p.$$

Its iterated fixpoint is

$$\begin{aligned} IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\ IFP^P \uparrow 1 &= IFP^P \uparrow 0 = WFM(P_3). \end{aligned}$$

P_3 is not dynamically stratified and assigns value undefined to p . So negative loops are resolved by assigning value undefined to the atom.

Let us consider the program P_5 :

$$p \leftarrow \sim q.$$

$$q \leftarrow \sim p.$$

Its iterated fixpoint is

$$\begin{aligned} IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\ IFP^P \uparrow 1 &= IFP^P \uparrow 0 = WFM(P_5). \end{aligned}$$

Thus, the depth of P_5 is 0 and the WFM of P_5 is given by

$$\begin{aligned} true(WFM(P_5)) &= \emptyset \\ undef(WFM(P_5)) &= \{p, q\} \\ false(WFM(P_5)) &= \emptyset. \end{aligned}$$

So $WFM(P_5)$ is three-valued and P_5 is not dynamically stratified.

Given its similarity with the T_P operator, it can be shown that $OpTrue_T^P$ reaches its fixpoint at most at the first infinite ordinal ω . IFP^P instead doesn't satisfy this property, as the next example shows.

Example 4 (Fixpoint of IFP^P beyond ω). Consider the following program inspired from [Hitzler and Seda, 2016, Program 2.6.5, page 58]:

$$\begin{aligned} p(0, 0). \\ p(Y, s(X)) \leftarrow r(Y), p(Y, X). \\ q(Y, s(X)) \leftarrow \sim p(Y, X). \\ r(0). \\ r(s(Y)) \leftarrow \sim q(Y, s(X)). \\ t \leftarrow \sim q(Y, s(X)). \end{aligned}$$

The ordinal powers of IFP^P are shown in Figure 1.4, where $s^n(0)$ is the term where the functor s is applied n times to 0. We need to reach the immediate successor of ω to compute the WFS of this program.

$$\begin{aligned}
IFP^P \uparrow 0 &= \langle \emptyset, \emptyset \rangle; \\
IFP^P \uparrow 1 &= \langle \{p(0, s^n(0)) | n \in \mathbb{N}\} \cup \{r(0)\}, \\
&\quad \{q(s^m(0), 0) | m \in \mathbb{N}\} \rangle; \\
IFP^P \uparrow 2 &= \langle \{p(0, s^n(0)) | n \in \mathbb{N}\} \cup \{r(0)\}, \\
&\quad \{q(0, s^n(0)) | n \in \mathbb{N}_1\} \cup \{q(s^m(0), 0) | m \in \mathbb{N}\} \rangle; \\
IFP^P \uparrow 3 &= \langle \{p(s^m(0), s^n(0)) | n \in \mathbb{N}, m \in \{0, 1\}\} \cup \\
&\quad \{r(0), r(s(0))\}, \\
&\quad \{q(0, s^n(0)) | n \in \mathbb{N}_1\} \cup \{q(s^m(0), 0) | m \in \mathbb{N}\} \rangle; \\
IFP^P \uparrow 4 &= \langle \{p(s^m(0), s^n(0)) | n \in \mathbb{N}, m \in \{0, 1\}\} \cup \\
&\quad \{r(0), r(s(0)), r(s(s(0)))\}, \\
&\quad \{q(0, s^n(0)), q(s(0), s^n(0)) | n \in \mathbb{N}_1\} \cup \\
&\quad \{q(s^m(0), 0) | m \in \mathbb{N}\} \rangle; \\
&\quad \dots \\
IFP^P \uparrow 2(i+1) &= \langle \{p(s^m(0), s^n(0)) | n \in \mathbb{N}, m \in \{0, \dots, i\}\} \cup \\
&\quad \{r(s^m(0)) | m \in \{0, \dots, i+1\}\}, \\
&\quad \{q(s^m(0), s^n(0)) | n \in \mathbb{N}_1, m \in \{0, \dots, i\}\} \cup \\
&\quad \{q(s^m(0), 0) | m \in \mathbb{N}\} \rangle; \\
&\quad \dots \\
IFP^P \uparrow \omega &= \langle \{p(s^m(0), s^n(0)) | n, m \in \mathbb{N}\} \cup \\
&\quad \{r(s^m(0)) | m \in \mathbb{N}\}, \\
&\quad \{q(s^m(0), s^n(0)) | n, m \in \mathbb{N}\} \rangle; \\
IFP^P \uparrow \omega + 1 &= \langle \{t\} \cup \{p(s^m(0), s^n(0)) | n, m \in \mathbb{N}\} \cup \\
&\quad \{r(s^m(0)) | m \in \mathbb{N}\}, \\
&\quad \{q(s^m(0), s^n(0)) | n, m \in \mathbb{N}\} \rangle;
\end{aligned}$$

Figure 1.4 Ordinal powers of IFP^P for the program of Example 4.

The following properties identify important subsets of programs.

Definition 4 (Acyclic, stratified and locally stratified programs).

- A level mapping for a program P is a function $|\cdot| : \mathcal{B}_P \rightarrow \mathbb{N}$ from ground atoms to natural numbers. For $a \in \mathcal{B}_P$, $|a|$ is the level of a . If $l = \neg a$ where $a \in \mathcal{B}_P$, we define $|l| = |a|$.
- A program T is called acyclic [Apt and Bezem, 1991] if there exists a level mapping such as, for every ground instance $a \leftarrow B$ of a clause of T , the level of a is greater than the level of each literal in B .
- A program T is called locally stratified if there exists a level mapping such as, for every ground instance $a \leftarrow B$ of a clause of T , the level of a is greater than the level of each negative literal in B and greater or equal than the level of each positive literals.
- A program T is called stratified if there exists a level mapping according to which the program is locally stratified and such that all the ground atoms for the same predicate can be assigned the same level.

The WFS for locally stratified programs enjoys the following property.

Theorem 1 (WFS for locally stratified programs [Van Gelder et al., 1991]). *If P is locally stratified, then it has a total WFM.*

Programs P_1 and P_2 of Example 5 are (locally) stratified while programs P_3 , P_4 , and P_5 are not (locally) stratified. Note that stratification is stronger than local stratification that is stronger than dynamic stratification.

SLG resolution [Chen and Warren, 1996] is a proof procedure that is sound and complete (under certain conditions) for the WFS. SLG uses *tabling*: it keeps a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, its answers are retrieved from the store rather than recomputing them. Besides saving time, tabling ensures termination for programs without function symbols. For a discussion of the termination properties of SLG in the general case, see [Riguzzi and Swift, 2014]. SLG resolution is implemented in the Prolog systems XSB [Swift and Warren, 2012], YAP [Santos Costa et al., 2012], and SWI-Prolog [Wielemaker et al., 2012].

1.4.3 Stable Model Semantics

The stable model semantics [Gelfond and Lifschitz, 1988] associates zero, one, or more two-valued models to a normal program.

Definition 5 (Reduction). *Given a normal program P and a two-valued interpretation I , the reduction P^I of P relative to I is obtained from $\text{ground}(P)$ by deleting*

1. *each rule that has a negative literal $\sim a$ such that $a \in I$*
2. *all negative literals in the body of the remaining rules.*

Thus, P^I is a program without negation as failure and has a unique least Herbrand model $\text{lh}_m(P^I)$.

Definition 6 (Stable model). *A two-valued interpretation I is a stable model or an answer set of a program P if $I = \text{lh}_m(P^I)$.*

The *stable model semantics* of a program P is the set of its stable models.

The relationship between the WFS and the stable model semantics is given by the following two theorems [Van Gelder et al., 1991].

Theorem 2 (WFS total model vs stable models). *If P has a total WFM, then that model is the unique stable model.*

Theorem 3 (WFS vs stable models). *The WFM of P is a subset of every stable model of P seen as a three-valued interpretation.*

Answer Set Programming (ASP) is a problem-solving paradigm based on the computation of the answer sets of a program.

Example 5 (Answer set computation).

Let us consider program P_1 of Example 2:

$b \leftarrow \sim a.$

$c \leftarrow \sim b.$

$c \leftarrow a.$

Its only answer set is $\{b\}$.

Let us consider program P_4 from [Przymusinski, 1989]

$b \leftarrow \sim a.$

$c \leftarrow \sim b.$

$c \leftarrow a, \sim p.$

$p \leftarrow \sim q.$

$q \leftarrow \sim p, b.$

The program has the answer sets $\{b, p\}$ and $\{b, q\}$ and $WFM(P_4) = \langle \{b\}, \{a, c\} \rangle$ is a subset of both seen as the three-valued interpretations $\langle \{b, p\}, \{a, c, q\} \rangle$ and $\langle \{b, q\}, \{a, c, p\} \rangle$

Let us consider program P_2 of Example 2:

$p \leftarrow p.$

Its only answer set is \emptyset .

Program P_3 of Example 2

$p \leftarrow \sim p.$

has no answer sets.

Program P_5 :

$p \leftarrow \sim q.$

$q \leftarrow \sim p.$

has the answer sets $\{p\}$ and $\{q\}$ and $WFM(P_5) = \langle \emptyset, \emptyset \rangle$ is a subset of both seen as the three-valued interpretations $\langle \{p\}, \{q\} \rangle$ and $\langle \{q\}, \{p\} \rangle$.

In general, loops through an odd number of negations may cause the program to have no answer set, while loops through an even number of negations may cause the program to have multiple answer sets. In this sense, the stable model semantics differs from the WFS that makes the atoms involved in the loops undefined in both cases.

DLV [Leone et al., 2006; Alviano et al., 2017], Smodels [Syrjänen and Niemelä, 2001], and Potassco [Gebser et al., 2011] are examples of systems for ASP.

1.5 Probability Theory

Probability theory provides a formal mathematical framework for dealing with uncertainty. The notions used in the book are briefly reviewed here; for a more complete treatment, please refer to textbooks such as [Ash and Doléans-Dade, 2000; Chow and Teicher, 2012].

Let W be a set called the *sample space*, whose elements are the *outcomes* of the random process we would like to model. For example, if the process is that of throwing a coin, the sample space can be $W^{coin} = \{h, t\}$ with h standing for heads and t for tails. If the process is that of throwing a die, $W^{die} = \{1, 2, 3, 4, 5, 6\}$. If we throw two coins, the sample space is $W^{2-coins} = \{(h, h), (h, t), (t, h), (t, t)\}$. If we throw an infinite sequence of coins, $W^{coins} = \{(o_1, o_2, \dots) | o_i \in \{h, t\}\}$. If we measure the position of an object along a single axis, $W^{pos.x} = \mathbb{R}$, on a plane $W^{pos.x.y} = \mathbb{R}^2$ and in the space $W^{pos.x.y.z} = \mathbb{R}^3$. If the object is limited to the unit interval, square, and cube, then $W^{unit.x} = [0, 1]$, $W^{unit.x.y} = [0, 1]^2$, and $W^{unit.x.y.z} = [0, 1]^3$.

Definition 7 (Algebra). *The set Ω of subsets of W is an algebra on the set W iff*

- $W \in \Omega$;
- Ω is closed under complementation, i.e., $\omega \in \Omega \rightarrow \omega^c = (\Omega \setminus \omega) \in \Omega$;
- Ω is closed under finite union, i.e., $\omega_1 \in \Omega, \omega_2 \in \Omega \rightarrow (\omega_1 \cup \omega_2) \in \Omega$

Definition 8 (σ -algebra). *The set Ω of subsets of W is a σ -algebra on the set W iff it is an algebra and*

- Ω is closed under countable union, i.e., if $\omega_i \in \Omega$ for $i = 1, 2, \dots$ then $\bigcup_i \omega_i \in \Omega$.

Definition 9 (Minimal σ -algebra). *Let \mathcal{A} be an arbitrary collection of subsets of W . The intersection of all σ -algebras containing all elements of \mathcal{A} is called the σ -algebra generated by \mathcal{A} , or the minimal σ -algebra containing \mathcal{A} . It is denoted by $\sigma(\mathcal{A})$.*

$\sigma(\mathcal{A})$ is such that $\Sigma \supseteq \sigma(\mathcal{A})$ whenever $\Sigma \supseteq \mathcal{A}$ and Σ is a σ -algebra. $\sigma(\mathcal{A})$ always exists and is unique [Chow and Teicher, 2012, page 7].

The elements of a σ -algebra Ω on W are called *measurable sets* or *events* and (W, Ω) is called a *measurable space*. When W is finite, Ω is usually the powerset of W . In general, however, not every subset of W need be present in Ω .

When throwing a coin, the set of events may be $\Omega^{coin} = \mathbb{P}(W^{coin})$ and $\{h\}$ is an event corresponding to the coin landing heads. When throwing

a die, an example of the set of events is $\Omega^{die} = \mathbb{P}(W^{die})$ and $\{1, 3, 5\}$ is an event, corresponding to obtaining an odd result. When throwing two coins, $(W^{2.coin}, \Omega^{2.coins})$ with $\Omega^{2.coins} = \mathbb{P}(W^{2.coins})$ is a measurable space.

When throwing an infinite sequence of coins, $\{(h, t, o_3, \dots) | o_i \in \{h, t\}\}$ may be an event, corresponding to obtaining head and tails in the first two throws. When measuring the position of an object on an axis and $W^{pos.x} = \mathbb{R}$, $\Omega^{pos.x}$ may be the Borel σ -algebra \mathcal{B} on the set of real numbers, the one generated, for example, by closed intervals $\{[a, b] : a, b \in \mathbb{R}\}$. Then $[-1, 1]$ may be an event, corresponding to observing the object in the $[-1, 1]$ interval. If the object is constrained to the unit interval $W^{uit.x}$, $\Omega^{unit.x}$ may be $\sigma(\{[a, b] : a, b \in [0, 1]\})$.

Definition 10 (Probability measure). *Given a measurable space (W, Ω) of subsets of W , a probability measure is a function $\mu : \Omega \rightarrow \mathbb{R}$ that satisfies the following axioms:*

μ -1 $\mu(\omega) \geq 0$ for all $\omega \in \Omega$;

μ -2 $\mu(W) = 1$;

μ -3 μ is countably additive, i.e., if $O = \{\omega_1, \omega_2, \dots\} \subseteq \Omega$ is a countable collection of pairwise disjoint sets, then $\mu(\bigcup_{\omega \in O} \omega) = \sum_i \mu(\omega_i)$.

(W, Ω, μ) is called a probability space.

We also consider finitely additive probability measures.

Definition 11 (Finitely additive probability measure). *Given a sample space W and an algebra Ω of subsets of W , a finitely additive probability measure is a function $\mu : \Omega \rightarrow \mathbb{R}$ that satisfies axioms $(\mu$ -1) and $(\mu$ -2) of Definition 10 and axiom*

μ -3 μ is finitely additive, i.e., $\omega_1 \cap \omega_2 = \emptyset \rightarrow \mu(\omega_1 \cup \omega_2) = \mu(\omega_1) + \mu(\omega_2)$ for all $\omega_1, \omega_2 \in \Omega$.

(W, Ω, μ) is called a finitely additive probability space.

Example 6 (Probability spaces). *When throwing a coin, $(W^{coin}, \Omega^{coin}, \mu^{coin})$ with $\mu^{coin}(\emptyset) = 0$, $\mu^{coin}(\{h\}) = 0.5$, $\mu^{coin}(\{t\}) = 0.5$, and $\mu^{coin}(\{h, t\}) = 1$ is a (finitely additive) probability space. When throwing a die, $(W^{die}, \Omega^{die}, \mu^{die})$ with $\mu^{die}(\omega) = |\omega| \cdot \frac{1}{6}$ is a (finitely additive) probability space.*

When throwing two coins, $(W^{2_coins}, \Omega^{2_coins}, \mu^{2_coins})$ with $\mu^{2_coins}(\omega) = |\omega| \cdot \frac{1}{36}$ is a (finitely additive) probability space. For the position of an object on an axis, $(W^{unit_x}, \Omega^{unit_x}, \mu^{unit_x})$ with $\mu^{unit_x}(I) = \int_I dx$ is a probability space. $(W^{pos_x}, \Omega^{pos_x}, \mu^{pos_x})$ with $\mu^{pos_x}(I) = \int_{I \cap [0,1]} dx$ is also a probability space.

Let (W, Ω, μ) be a probability space and (S, Σ) a measurable space. A function $X : W \rightarrow S$ is said to be *measurable* if the preimage of σ under X is in Ω for every $\sigma \in \Sigma$, i.e.,

$$X^{-1}(\sigma) = \{w \in W \mid X(w) \in \sigma\} \in \Omega, \quad \forall \sigma \in \Sigma.$$

Definition 12 (Random variable). *Let (W, Ω, μ) be a probability space and let (S, Σ) be a measurable space. A measurable function $X : W \rightarrow S$ is a random variable. We call the elements of S the values of X . With $P(X \in \sigma)$ for $\sigma \in \Sigma$, we indicate the probability that random variable X has value in σ , defined as $\mu(X^{-1}(\sigma))$.*

If Σ is finite or countable, X is a discrete random variable. If Σ is uncountable, then X is a continuous random variable.

We indicate random variables with Roman uppercase letters X, Y, \dots and the values with Roman lowercase letters x, y, \dots

When $(W, \Omega) = (S, \Sigma)$, X is often the identity function and $P(X = \omega) = \mu(\omega)$.

In the discrete case, the values of $P(X \in \{x\})$ for all $x \in S$ define the *probability distribution* of random variable X , often abbreviated as $P(X=x)$ and $P(x)$. We indicate the probability distribution for random variable X with $P(X)$.

Example 7 (Discrete random variables). *An example of a discrete random variable X for the probability space $(W^{coin}, \Omega^{coin}, \mu^{coin})$ is the identity function and the probability distribution is $P(X=h) = 0.5, P(X=t) = 0.5$. When throwing a die, a discrete random variable X can be the identity and $P(X=n) = \frac{1}{6}$ for all $n \in \{1, 2, 3, 4, 5, 6\}$. Another discrete random variable E for a die can represent whether the outcome was even or odd with $(S, \Sigma) = (\{e, o\}, \mathbb{P}(S))$ and $E: W^{die} \rightarrow S$ defined as $E = \{1 \rightarrow o, 2 \rightarrow e, 3 \rightarrow o, 4 \rightarrow e, 5 \rightarrow o, 6 \rightarrow e\}$. The probability distribution is then $P(E = e) = \mu^{die}(\{2, 4, 6\}) = \frac{1}{6} \times 3 = \frac{1}{2}$ and $P(E = o) = \mu^{die}(\{1, 3, 5\}) = \frac{1}{6} \times 3 = \frac{1}{2}$.*

For the probability space $(W^{2\text{-coins}}, \Omega^{2\text{-coins}}, \mu^{2\text{-coins}})$, a discrete random variable may be the function $X : W^{2\text{-coins}} \rightarrow W^{\text{coin}}$ defined as

$$X(\{c_1, c_2\}) = c_1,$$

i.e., the function returning the outcome of the first coin. For value $h \in W^{\text{coin}}$, we have

$$X^{-1}(\{h\}) = \{(h, h), (h, t)\}$$

and

$$P(X = h) = \mu^{2\text{-coins}}(\{(h, h), (h, t)\}) = 0.5.$$

In the continuous case, we define the cumulative distribution and the probability density.

Definition 13 (Cumulative distribution and probability density). *The cumulative distribution of a random variable $X : (W, \Omega) \rightarrow (\mathbb{R}, \mathcal{B})$ is the function $F(x) : \mathbb{R} \rightarrow [0, 1]$ defined as*

$$F(x) = P(X \in \{t | t \leq x\}).$$

We write $P(X \in \{t | t \leq x\})$ also as $P(X \leq x)$. The probability density of X is a function $p(x)$ such that

$$P(X \in A) = \int_A p(x) dx$$

for any measurable set $A \in \mathcal{B}$.

It holds that:

$$F(x) = \int_{-\infty}^x p(t) dt$$

$$p(x) = \frac{dF(x)}{dx}$$

$$P(X \in [a, b]) = F(b) - F(a) = \int_a^b p(x) dx$$

A discrete random variable is described by giving its probability distribution $P(X)$ while a continuous random variable is described by giving its cumulative distribution $F(x)$ or probability density $p(x)$.

Example 8 (Continuous random variables). For the probability space of an object in the unit interval $(W^{unit-x}, \Omega^{unit-x}, \mu^{unit-x})$, the identity X is a continuous random variable with cumulative distribution and density

$$F(x) = \int_0^x dt = x$$

$$p(x) = 1$$

for $x \in [0, 1]$. For probability space $(W^{pos-x}, \Omega^{pos-x}, \mu^{pos-x})$, the identity X is a continuous random variable with cumulative distribution and density

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \in [0, 1] \\ 1 & \text{if } x > 1 \end{cases}$$

$$p(x) = \begin{cases} 1 & \text{if } x \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

This is an example of an uniform density. In general, the uniform density in the interval $[a, b]$ is

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

Another notable density is the normal or Gaussian density, with

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ and σ are parameters denoting the mean and the standard deviation (σ^2 is the variance). We use $\mathcal{N}(\mu, \sigma)$ to indicate a normal density. Examples of Gaussian densities for various values of the parameters are shown in Figure 1.5.

When the values of a random variable are numeric, we can compute its *expected value* or *expectation*, which intuitively is the average of the values obtained by repeating infinitely often the experiment it represents. The expectation of a discrete variable X is

$$\mathbf{E}(X) = \sum_x xP(x)$$

while the expectation of a continuous variable X with domain \mathbb{R} is

$$\mathbf{E}(X) = \int_{-\infty}^{+\infty} xp(x)dx.$$

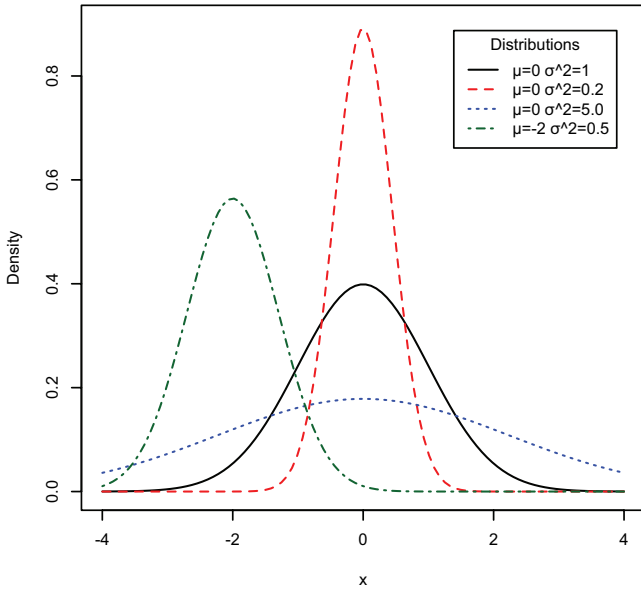


Figure 1.5 Gaussian densities.

Multiple random variables can be defined on the same space (W, Ω, μ) , especially when $(W, \Omega) \neq (S, \Sigma)$. Suppose two random variables X and Y are defined on the probability space (W, Ω, μ) and measurable spaces (S_1, Σ_1) and (S_2, Σ_2) , respectively. We can define the *joint probability* $P(X \in \sigma_1, Y \in \sigma_2)$ as $\mu(X^{-1}(\sigma_1) \cap Y^{-1}(\sigma_2))$ for $\sigma_1 \in \Sigma_1$ and $\sigma_2 \in \Sigma_2$. Since X and Y are random variables and Ω is a σ -algebra, then $X^{-1}(\sigma_1) \cap Y^{-1}(\sigma_2) \in \Omega$ and the joint probability is well defined. If X and Y are discrete, the values of $P(X = \{x\}, Y = \{y\})$ for all $x \in S_1$ and $y \in S_2$ define the *joint probability distribution* of X and Y often abbreviated as $P(X = x, Y = y)$ and $P(x, y)$. We call $P(X, Y)$ the joint probability distribution of X and Y . If X and Y are continuous, we can define the *joint cumulative distribution* $F(x, y)$ as $P(X \in \{t|t \leq x\}, Y \in \{t|t \leq y\})$ and *joint probability density* as the function $p(x, y)$ such that $P(X \in A, Y \in B) = \int_A \int_B p(x, y) dx dy$.

If X is discrete and Y continuous, we can still define the joint cumulative distribution $F(x, y)$ as $P(X = x, Y \in \{t|t \leq y\})$ and the joint probability density as the function $p(x, y)$ such that $P(X = x, Y \in B) = \int_B p(x, y) dy$.

Example 9 (Joint distributions). *When throwing two coins, we can define two random variables X_1 and X_2 as $X_1((c_1, c_2)) = c_1$ and $X_2((c_1, c_2)) =$*

c_2 : the first indicates the outcome of the first coin and the second of the second. The joint probability distribution is $P(X_1 = x_1, X_2 = x_2) = \frac{1}{4}$ for all $x_1, x_2 \in \{h, t\}$.

For the position of an object on a plane, we can have random variable X for its position on the X -axis and Y on the Y -axis. An example of joint probability density is the multivariate normal or Gaussian density

$$p(\mathbf{x}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k \det \boldsymbol{\Sigma}}} \quad (1.2)$$

where \mathbf{x} is a real two-dimensional column vector, $\boldsymbol{\mu}$ is a real two-dimensional column vector; the mean, $\boldsymbol{\Sigma}$ is a 2×2 symmetric positive-definite¹ matrix called covariance matrix, and $\det \boldsymbol{\Sigma}$ is the determinant of $\boldsymbol{\Sigma}$.

A bivariate Gaussian density with parameters $\boldsymbol{\mu} = [0, 0]^T$ and

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}$$

is shown in Figure 1.6.

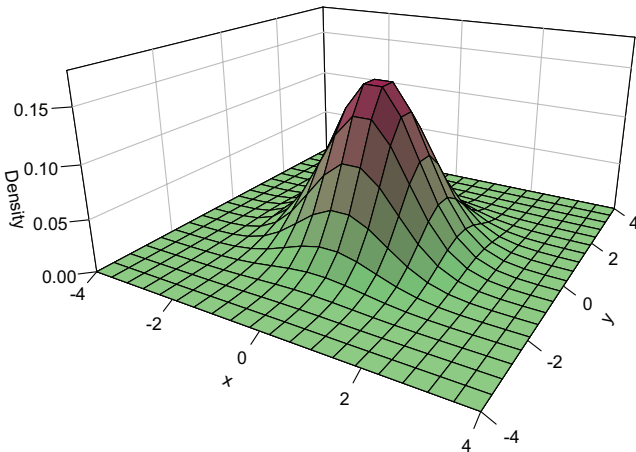


Figure 1.6 Bivariate Gaussian density.

¹All its eigenvalues are positive.

Definition 14 (Product σ -algebra and product space). *Given two measurable spaces (W_1, Ω_1) and (W_2, Ω_2) , define the product σ -algebra $\Omega_1 \otimes \Omega_2$ as*

$$\Omega_1 \otimes \Omega_2 = \sigma(\{\omega_1 \times \omega_2 \mid \omega_1 \in \Omega_1, \omega_2 \in \Omega_2\}).$$

Note that $\Omega_1 \otimes \Omega_2$ differs from the Cartesian product $\Omega_1 \times \Omega_2$ because it is the minimal σ -algebra generated by all possible Cartesian products of couples of elements from Ω_1 and Ω_2 .

Then define the product space $(W_1, \Omega_1) \times (W_2, \Omega_2)$ as

$$(W_1, \Omega_1) \times (W_2, \Omega_2) = (W_1 \times W_2, \Omega_1 \otimes \Omega_2).$$

Given two random variables X and Y that are defined on the probability space (W, Ω, μ) and measurable spaces (S_1, Σ_1) and (S_2, Σ_2) , respectively, we can define the function XY such that $XY(w) = (X(w), Y(w))$. Such a function is a random variable defined on the same probability space and the product measurable space $(S_1, \Sigma_1) \times (S_2, \Sigma_2)$ [Chow and Teicher, 2012, Theorem 1, page 13]. So, for example, two discrete random variables X and Y with values x_i and y_j with their joint distribution $P(X, Y)$ can be interpreted as a random variable XY with values (x_i, y_j) and distribution $P((X, Y))$. The results in the following thus apply also by replacing a random variable with a vector (or set) of random variables. We usually denote vectors of random variables with boldface capital letters such as $\mathbf{X}, \mathbf{Y}, \dots$ and vectors of values with boldface lowercase letters such as $\mathbf{x}, \mathbf{y}, \dots$.

Definition 15 (Conditional probability). *Given two discrete random variables X and Y and two values x and y for them, if $P(x) > 0$, we can define the conditional probability $P(x|y)$ as*

$$P(x|y) = \frac{P(x, y)}{P(y)}$$

If $P(y) = 0$, then $P(x|y)$ is not defined.

$P(x|y)$ provides the probability distribution of variable X given that random variable Y was observed having value y .

From this, we get the important *product rule*

$$P(x, y) = P(x|y)P(y)$$

expressing a joint distribution in terms of a conditional distribution.

Moreover, x and y can be exchanged obtaining $P(x, y) = P(y|x)P(x)$, so by equating the two expressions, we find *Bayes' theorem*

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}.$$

Now consider two discrete random variables X and Y , with X having a finite set of values $\{x_1, \dots, x_n\}$, and let y be a value of Y . The sets

$$X^{-1}(\{x_1\}) \cap Y^{-1}(\{y\}), \dots, X^{-1}(\{x_n\}) \cap Y^{-1}(\{y\})$$

are mutually exclusive because X is a function and

$$Y^{-1}(y) = X^{-1}(\{x_1\}) \cap Y^{-1}(\{y\}) \cup \dots \cup X^{-1}(\{x_n\}) \cap Y^{-1}(\{y\}).$$

Given the fact that probability measures are additive, then

$$P(X \in \{x_1, \dots, x_n\}, y) = P(y) = \sum_{i=1}^n P(x_i, y)$$

This is the *sum rule* of probability theory, often expressed as

$$P(y) = \sum_x P(x, y)$$

The sum rule eliminates a variable from a joint distribution and we also say that we *sum out* the variable and that we *marginalize* the joint distribution.

For continuous random variables, we can similarly define the *conditional density* $p(X|y)$ as

$$p(x|y) = \frac{p(x, y)}{p(y)}$$

when $p(y) > 0$, and get the following versions of the product and sum rules

$$\begin{aligned} p(x, y) &= p(x|y)p(y) \\ p(y) &= \int_{-\infty}^{\infty} p(x, y) dx \end{aligned}$$

We can also define *conditional expectations* as

$$\begin{aligned} \mathbf{E}(X|y) &= \sum_x xP(x|y) \\ \mathbf{E}(X|y) &= \int_{-\infty}^{+\infty} xp(x|y)dx. \end{aligned}$$

for X a discrete or continuous variable, respectively.

For continuous random variables, we may have evidence on Y in the form of the measurement of a value y for it. In many practical cases, such as the Gaussian distribution, $P(y) = 0$ for all values y of Y , so the conditional probability $P(X \in \omega_1, y)$ is not defined. This is known as the *Borel–Kolmogorov paradox* [Gyenis et al., 2017].

In some cases, it can be solved by defining the conditional probability as

$$P(X \in \omega_1|y) = \lim_{dv \rightarrow 0} \frac{P(X \in \omega_1, Y \in [y - dv/2, y + dv/2])}{P(Y \in [y - dv/2, y + dv/2])}.$$

However, the limit is not always defined.

Definition 16 (Independence and conditional independence). *Two random variables X and Y are independent, indicated with $I(X, Y, \emptyset)$ iff*

$$P(x|y) = P(x) \text{ whenever } P(y) > 0$$

Two random variables X and Y are conditionally independent given Z , indicated with $I(X, Y, Z)$ iff

$$P(x|y, z) = P(x|z) \text{ whenever } P(y, z) > 0$$

1.6 Probabilistic Graphical Models

It is often convenient to describe a domain using a set of random variables. For example, a home intrusion detection system can be described with the random variables

- Earthquake E , which assumes value true (t) if an earthquake occurred and false (f) otherwise;
- Burglary B , which assumes value true (t) if a burglary occurred and false (f) otherwise;
- Alarm A , which assumes value true (t) if the alarm went off and false (f) otherwise;
- Neighbor call N , which assumes value true (t) if the neighbor called and false (f) otherwise.

These variables may describe the situation of the system at a particular point in time, such as last night. We would like to answer the questions such as

- What is the probability of a burglary? (compute $P(B = t)$, belief computation)
- What is the probability of a burglary given that the neighbor called? (compute $P(B = t|N = t)$, belief updating)
- What is the probability of a burglary given that there was an earthquake and the neighbor called? (compute $P(B = t|N = t, E = t)$, belief updating)
- What is the probability of a burglary and of the alarm ringing given that there was an earthquake and the neighbor called? (compute $P(A = t, B = t|N = t, E = t)$, belief updating)
- What is the most likely value for burglary given that the neighbor called? ($\arg \max_b P(b|N = t)$, belief revision).

When assigning a causal meaning to the variables, the problems are also called

- Diagnosis: computing $P(\text{cause}|\text{symptom})$.
- Prediction: computing $P(\text{symptom}|\text{cause})$.

Moreover, another inference problem is

- Classification: computing $\arg \max_{\text{class}} P(\text{class}|\text{data})$.

In general, we want to compute the probability $P(\mathbf{q}|\mathbf{e})$ of a *query* \mathbf{q} (assignment of values to a set of variables \mathbf{Q}) given the evidence \mathbf{e} (assignment of values to a set of variables \mathbf{E}). This problem is called *inference*. If \mathbf{X} denotes the set of all variables describing the domain and we know the joint probability distribution $P(\mathbf{X})$, i.e., we know $P(\mathbf{x})$ for all \mathbf{x} , we can answer all types of queries using the definition of conditional probability and the sum rule:

$$P(\mathbf{q}|\mathbf{e}) = \frac{P(\mathbf{q}, \mathbf{e})}{P(\mathbf{e})} = \frac{\sum_{\mathbf{y}, \mathbf{Y}=\mathbf{X} \setminus \mathbf{Q} \setminus \mathbf{E}} P(\mathbf{y}, \mathbf{q}, \mathbf{e})}{\sum_{\mathbf{z}, \mathbf{Z}=\mathbf{X} \setminus \mathbf{E}} P(\mathbf{z}, \mathbf{e})}$$

However, if we have n binary variables ($|\mathbf{X}| = n$), knowing the joint probability distribution requires storing $O(2^n)$ different values. Even if we had the space to store all the 2^n different values, computing $P(\mathbf{q}|\mathbf{e})$ would require $O(2^n)$ operations. Therefore, this approach is impractical for real-world problems and a different solution must be adopted.

First note that if $\mathbf{X} = \{X_1, \dots, X_n\}$, a value of \mathbf{X} is a tuple (x_1, \dots, x_n) also called a *joint event* and we can write

$$\begin{aligned}
P(\mathbf{x}) &= P(x_1, \dots, x_n) = \\
&P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1}, \dots, x_1) = \\
&\dots \\
&P(x_n | x_{n-1}, \dots, x_1) \dots P(x_2 | x_1) P(x_1) = \quad (1.3) \\
&\prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1)
\end{aligned}$$

where Equation (1.3) is obtained by repeated application of the product rule. This formula expresses the so-called *chain rule*.

Now if we knew, for each variable X_i , a subset \mathbf{Pa}_i of $\{X_{i-1}, \dots, X_1\}$ such that X_i is conditionally independent of $\{X_{i-1}, \dots, X_1\} \setminus \mathbf{Pa}_i$ given \mathbf{Pa}_i , i.e.,

$$P(x_i | x_{i-1}, \dots, x_1) = P(x_i | \mathbf{pa}_i) \text{ whenever } P(x_{i-1}, \dots, x_1) > 0,$$

then we could write

$$\begin{aligned}
P(\mathbf{x}) &= P(x_1, \dots, x_n) = \\
&P(x_n | x_{n-1}, \dots, x_1) \dots P(x_2 | x_1) P(x_1) = \\
&P(x_n | \mathbf{pa}_n) \dots P(x_2 | \mathbf{pa}_1) P(x_1 | \mathbf{pa}_1) = \\
&\prod_{i=1}^n P(x_i | \mathbf{pa}_i)
\end{aligned}$$

Therefore, in order to compute $P(\mathbf{x})$, we have to store $P(x_i | \mathbf{pa}_i)$ for all values x_i and \mathbf{pa}_i . The set of values $P(x_i | \mathbf{pa}_i)$ is called the Conditional Probability Table (CPT) of variable X_i . If \mathbf{Pa}_i is much smaller than $\{X_{i-1}, \dots, X_1\}$, then we have huge savings. For example, if k is the maximum size of \mathbf{Pa}_i , then the storage requirements are $O(n2^k)$ instead of $O(2^n)$. So it is important to take into account independencies among the variables as they enable much faster inference. One way to do this is by means of *graphical models* that are graph structures that represent independencies.

An example of a graphical model is a Bayesian Network (BN) [Pearl, 1988] that represents a set of variables with a directed graph with a node per variable and an edge from X_j to X_i only if $X_j \in \mathbf{Pa}_i$. The variables in \mathbf{Pa}_i are in fact also called the *parents* of X_i and X_i is called a *child* of every node in \mathbf{Pa}_i . Given that $\mathbf{Pa}_i \subseteq \{X_{i-1}, \dots, X_1\}$, the parents of X_i will always have an index lower than i and the ordering $\langle X_1, \dots, X_n \rangle$ is a topological sort of the graph that is therefore acyclic. A BN together with the set of CPTs $P(x_i | \mathbf{pa}_i)$ defines a joint probability distribution.

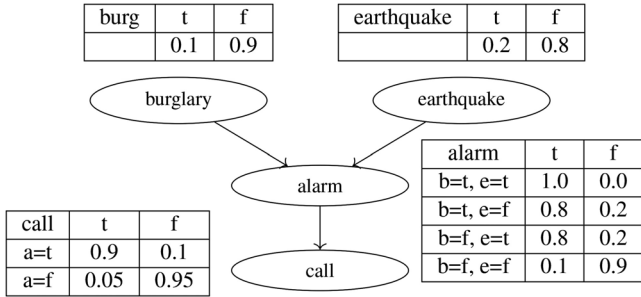


Figure 1.7 Example of a Bayesian network.

Example 10 (Alarm – BN). For our intrusion detection system and the variable order $\langle E, B, A, N \rangle$, we can identify the following independencies

$$\begin{aligned}
 P(e) &= P(e) \\
 P(b|e) &= P(b) \\
 P(a|b, e) &= P(a|b, e) \\
 P(n|a, b, e) &= P(n|a)
 \end{aligned}$$

which result in the BN of Figure 1.7 that also shows the CPTs.

When a CPT contains only the values 0.0 and 1.0, the dependency of the child from its parents is *deterministic* and the CPT encodes a function: given the values of the parents, the value of the child is completely determined. For example, if the variables are Boolean, a *deterministic CPT* can encode the AND or the OR Boolean function.

The concepts of ancestors of a node and of descendants of a node are defined as the transitive closure of the parent and child relationships: if there is a directed path from X_i to X_j , then X_i is an *ancestor* of X_j and X_j is a *descendant* of X_i . Let $ANC(X)$ ($DESC(X)$) be the set of ancestors (descendants) of X .

From the definition of BN, we know that, given its parents, a variable is independent of its other ancestors. However, BNs allow reading other independence relationship using the notion of d-separation.

Definition 17 (d-separation [Murphy, 2012]). An undirected path P in a BN is a path connecting two nodes not considering edge directions.

An undirected path P is d-separated by a set of nodes C iff at least one of the following conditions hold:

- P contains a chain, $S \rightarrow M \rightarrow T$ or $S \leftarrow M \leftarrow T$; where $M \in \mathbf{C}$;
- P contains a fork, $S \leftarrow M \rightarrow T$, where $M \in \mathbf{C}$
- P contains a collider $S \rightarrow M \leftarrow T$, where $M \notin \mathbf{C}$ and $\forall X \in \text{DESC}(M) : X \notin \mathbf{C}$.

Two sets of random variables \mathbf{A} and \mathbf{B} are d-separated given a set \mathbf{C} if and only if each undirected path from every node $A \in \mathbf{A}$ to every node $B \in \mathbf{B}$ is d-separated by \mathbf{C} .

It is possible to prove that \mathbf{A} is independent from \mathbf{B} given \mathbf{C} iff \mathbf{A} is d-separated from \mathbf{B} given \mathbf{C} , so d-separation and conditional independence are equivalent.

The set of parents, children, and other children of the parents of a variable d-separates it from the other variables, so forms a sufficient set of nodes to make the variable independent from all the others. Such a set is called a *Markov blanket* and is shown in Figure 1.8.

Graphical models are a way to represent a factorization of a joint probability distribution. BNs represent a factorization in terms of conditional probabilities. In general, a *factorized model* takes the form

$$P(\mathbf{x}) = \frac{\prod_{i=1}^m \phi_i(\mathbf{x}_i)}{Z}$$

where each $\phi_i(\mathbf{x}_i)$ for $i = 1, \dots, m$ is a *potential* or *factor*, a function taking non-negative values on a subset \mathbf{X}_i of \mathbf{X} . Factors are indicated with ϕ, ψ, \dots

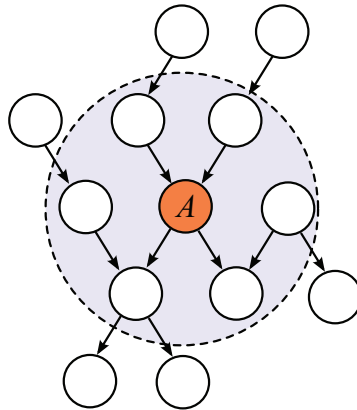


Figure 1.8 Markov blanket. Figure from https://commons.wikimedia.org/wiki/File:Diagram_of_a_Markov_blanket.svg.

Since the numerator of the fraction may not lead to a probability distribution (the sum of all possible values at the numerator may not be one), the denominator Z is added that makes the expression a probability distribution since it is defined as

$$Z = \sum_{\mathbf{x}} \prod_i \phi_i(\mathbf{x}_i)$$

i.e., it is exactly the sum of all possible values of the numerator. Z is thus a *normalizing constant* and is also called *partition function*.

A potential $\phi_i(\mathbf{x}_i)$ can be seen as a table that, for each possible combination of values of the variables in \mathbf{X}_i , returns a non-negative number. Since all potentials are non-negative, $P(\mathbf{x}) \geq 0$ for all \mathbf{x} . Potentials influence the distribution in this way: all other things being equal, a larger value of a potential for a combination of values of the variables in its domain corresponds to a larger probability value of the instantiation of \mathbf{X} .

An example of a potential for a university domain is a function defined over the random variables Intelligent and GoodMarks expressing the fact that a student is intelligent and that he gets good marks, respectively. The potential may be represented by the table

Intelligent	GoodMarks	$\phi_i(I, G)$
false	false	4.5
false	true	4.5
true	false	1.0
true	true	4.5

where the configuration where the student is intelligent but he does not get good marks has a lower value than the other configurations.

A BN can be seen as defined a factorized model with a potential for each *family* of nodes (a node and its parents) defined as $\phi_i(x_i, \mathbf{pa}_i) = P(x_i | \mathbf{pa}_i)$. It is easy to see that in this case $Z = 1$.

If all the potentials are strictly positive, we can replace each feature $\phi_i(\mathbf{x}_i)$ with the exponential $\exp(w_i f_i(\mathbf{x}_i))$ with w_i a real number called *weight* and $f_i(\mathbf{x}_i)$ a function from the values of \mathbf{X}_i to the reals (often only to $\{0, 1\}$) called *feature*. If the potentials are strictly positive, this reparameterization is always possible. In this case, the factorized model becomes:

$$P(\mathbf{x}) = \frac{\exp(\sum_i w_i f_i(\mathbf{x}_i))}{Z}$$

$$Z = \sum_{\mathbf{x}} \exp(\sum_i w_i f_i(\mathbf{x}_i))$$

This is also called a *log-linear model* because the logarithm of the joint is a linear function of the features. An example of a feature corresponding to the example potential above is

$$f_i(\text{Intelligent}, \text{GoodMarks}) = \begin{cases} 1 & \text{if } \neg\text{Intelligent} \vee \text{GoodMarks} \\ 0 & \text{otherwise} \end{cases}$$

If $w_i = 1.5$, then $\phi_i(i, g) = \exp(w_i f_i(i, g))$ for all values i, g for random variables I, G in our university example.

A Markov Network (MN) or Markov random field [Pearl, 1988] is an undirected graph that represents a factorized model. An MN has a node for each variable and each couple of nodes that appear together in the domain of a potential are connected by an edge. In other words, the nodes in the domain of each potential form a *clique*, a fully-connected subset of the nodes.

Example 11 (University – MN). *An MN for a university domain is shown in Figure 1.9. The network contains the example potential above plus a potential involving the three variables GoodMarks, CouDifficulty, and TeachAbility.*

As BNs, MNs also allow reading off independencies from the graph. In an MN where $P(\mathbf{x}) > 0$ for all \mathbf{x} (a strictly positive distribution), two sets of random variables \mathbf{A} and \mathbf{B} are independent given a set \mathbf{C} if and only if each path from every node $A \in \mathbf{A}$ to every node $B \in \mathbf{B}$ passes through an element of \mathbf{C} [Pearl, 1988]. Then the Markov blanket of a variable is the set of its neighbors. So reading independencies from the graph is much easier than for BNs.

MNs and BNs can each represent independencies that the other cannot represent [Pearl, 1988], so their expressive power is not comparable. MNs have the advantage that the potentials/features can be defined more freely, because they do not have to respect the normalization that conditional probabilities must respect. On the other hand, parameters in an MN are difficult

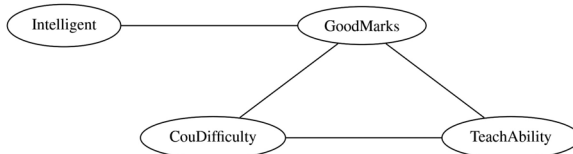


Figure 1.9 Example of a Markov network.

to interpret, because their influence on the joint distribution depends on the whole set of potentials, while in a BN, the parameters are conditional probabilities, so they are easier to interpret and to estimate.

Given a BN, an MN representing the same joint distribution can be obtained by *moralizing* the graph: adding edges between all pairs of nodes that have a common child (marrying the parents), and then making all edges in the graph undirected. In this way, each family $\{X_i, \mathbf{Pa}_i\}$ of the BN forms a clique in the MN associated with the potential $\phi_i(x_i, \mathbf{pa}_i) = P(x_i|\mathbf{pa}_i)$.

Given an MN, an equivalent BN can be obtained containing a node for each variable plus a Boolean node F_i for each potential ϕ_i . Edges are then added so that a potential node has all the nodes in its scope as parents. The BN equivalent to the MN of Figure 1.9 is shown in Figure 1.10.

The CPT for node F_i is

$$P(F_i = 1|\mathbf{x}_i) = \frac{\phi_i(\mathbf{x}_i)}{C_i}$$

where C_i is a constant depending on the potential that ensures that the values are probabilities, i.e., that they are in $[0, 1]$. It can be chosen freely provided that $C_i \geq \max_{\mathbf{x}_i} \phi_i(\mathbf{x}_i)$, for example, it can be chosen as $C_i = \max_{\mathbf{x}_i} \phi_i(\mathbf{x}_i)$ or $C_i = \sum_{\mathbf{x}_i} \phi_i(\mathbf{x}_i)$. The CPT for each variable node assigns uniform probability to its values, i.e., $P(x_j) = \frac{1}{k_j}$ where k_j is the number of values of X_j .

Then the joint conditional distribution $P(\mathbf{x}|\mathbf{F} = \mathbf{1})$ of the BN is equal to the joint of the MN. In fact

$$P(\mathbf{x}, \mathbf{F} = \mathbf{1}) = \prod_i P(F_i = 1|\mathbf{x}_i) \prod_j P(x_j) = \prod_i \frac{\phi_i(\mathbf{x}_i)}{C_i} \prod_j \frac{1}{k_j} = \frac{\prod_i \phi_i(\mathbf{x}_i)}{\prod_i C_i} \prod_j \frac{1}{k_j}$$

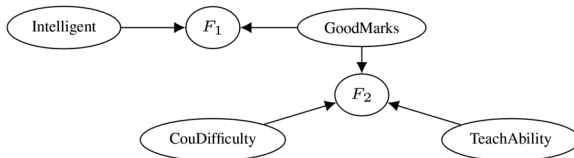


Figure 1.10 Bayesian network equivalent to the Markov network of Figure 1.9.

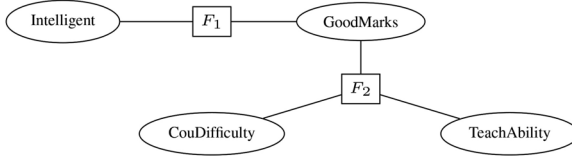


Figure 1.11 Example of a factor graph.

and

$$\begin{aligned}
 P(\mathbf{x}|\mathbf{F} = \mathbf{1}) &= \frac{P(\mathbf{x}, \mathbf{F} = \mathbf{1})}{P(\mathbf{F} = \mathbf{1})} = \frac{P(\mathbf{x}, \mathbf{F} = \mathbf{1})}{\sum_{\mathbf{x}'} P(\mathbf{x}', \mathbf{F} = \mathbf{1})} = \\
 &= \frac{\prod_i \phi_i(\mathbf{x}_i)}{\prod_i C_i} \prod_j \frac{1}{k_j} = \\
 &= \frac{\sum_{\mathbf{x}'} \prod_i \phi_i(\mathbf{x}'_i)}{\prod_i C_i} \prod_j \frac{1}{k_j} = \\
 &= \frac{\frac{1}{\prod_i C_i} \prod_j \frac{1}{k_j} \prod_i \phi_i(\mathbf{x}_i)}{\frac{1}{\prod_i C_i} \prod_j \frac{1}{k_j} \sum_{\mathbf{x}'} \prod_i \phi_i(\mathbf{x}'_i)} = \frac{\prod_i \phi_i(\mathbf{x}_i)}{Z}
 \end{aligned}$$

So, a query \mathbf{q} given evidence \mathbf{e} to the MN can be answered using the equivalent BN by computing $P(\mathbf{q}|\mathbf{e}, \mathbf{F} = \mathbf{1})$.

The equivalent BN encodes the same conditional independencies as the MN provided that the set of condition nodes is extended with the set of factor nodes, so an independence $I(\mathbf{A}, \mathbf{B}, \mathbf{C})$ holding in the MN will hold in the equivalent BN in the form $I(\mathbf{A}, \mathbf{B}, \mathbf{C} \cup \mathbf{F})$. For example, a couple of nodes from the scope \mathbf{X}_i of a factor given the factor node F_i cannot be made independent no matter what other nodes are added to the condition set because of d-separation, just as in the MN where they form a clique.

A third type of graphical model is the Factor Graph (FG) which can represent general factorized models. An FG is undirected and bipartite, i.e., its nodes are divided into two disjoint sets, the set of variables and the set of factors, and the edges always have an endpoint in a set and the other in the other set. A factor corresponds to a potential in the model. An FG contains an edge connecting each factor with all the variables in its domain. So from an FG, one can immediately read the factorization of the model. An FG representing the factorized model of Figure 1.9 is shown in Figure 1.11.

FGs are specially useful for expressing inference algorithms, in particular the formulas for belief propagation are simpler to express for FGs than for BNs and MNs.

2

Probabilistic Logic Programming Languages

Various approaches have been proposed for combining logic programming with probability theory. They can be broadly classified into two categories: those based on the *Distribution Semantics* (DS) [Sato, 1995] and those that follow a *Knowledge Base Model Construction* (KBMC) approach.

For languages in the first category, a probabilistic logic program without function symbols defines a probability distribution over normal logic programs (termed *worlds*). To define the probability of a query, this distribution is extended to a joint distribution of the query and the worlds and the probability of the query is obtained from the joint distribution by marginalization, i.e., by summing out the worlds. For probabilistic logic programs with function symbols, the definition is more complex, see Chapter 3.

The distribution over programs is defined by encoding random choices for clauses. Each choice generates an alternative version of the clause and the set of choices is associated with a probability distribution. The various languages that follow the DS differ in how the choices are encoded. In all languages, however, choices are independent from each other.

In the KBMC approach, instead, a probabilistic logic program is a compact way of encoding a large graphical model, either a BN or MN. In the KBMC approach, the semantics of a program is defined by the method for building the graphical model from the program.

2.1 Languages with the Distribution Semantics

The languages following DS differ in how they encode choices for clauses, and how the probabilities for these choices are stated. As will be shown in Section 2.4, they all have the same expressive power. This fact shows that the differences in the languages are syntactic, and also justifies speaking of *the DS*.

2.1.1 Logic Programs with Annotated Disjunctions

In Logic Programs with Annotated Disjunctions (LPADs) [Vennekens et al., 2004], the alternatives are expressed by means of annotated disjunctive heads of clauses. An *annotated disjunctive clause* C_i has the form

$$h_{i1} : \Pi_{i1} ; \dots ; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \dots, b_{im_i}$$

where h_{i1}, \dots, h_{in_i} are logical atoms, b_{i1}, \dots, b_{im_i} are logical literals, and $\Pi_{i1}, \dots, \Pi_{in_i}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} = 1$. An LPAD is a finite set of annotated disjunctive clauses.

Each world is obtained by selecting one atom from the head of each grounding of each annotated disjunctive clause.

Example 12 (Medical symptoms – LPAD). *The following LPAD models the appearance of medical symptoms as a consequence of disease. A person may sneeze if he has the flu or if he has hay fever:*

$$\begin{aligned} & \text{sneezing}(X) : 0.7 ; \text{null} : 0.3 \leftarrow \text{flu}(X). \\ & \text{sneezing}(X) : 0.8 ; \text{null} : 0.2 \leftarrow \text{hay_fever}(X). \\ & \text{flu}(\text{bob}). \\ & \text{hay_fever}(\text{bob}). \end{aligned}$$

*The first clause can be read as: if X has the flu, then X sneezes with probability 0.7 and nothing happens with probability 0.3. Similarly, the second clause can be read as: if X has hay fever, then X sneezes with probability 0.8 and nothing happens with probability 0.2. Here, and for the other languages based on the distribution semantics, the atom *null* does not appear in the body of any clause and is used to represent an alternative in which no atom is selected. It can also be omitted obtaining*

$$\begin{aligned} & \text{sneezing}(X) : 0.7 \leftarrow \text{flu}(X). \\ & \text{sneezing}(X) : 0.8 \leftarrow \text{hay_fever}(X). \\ & \text{flu}(\text{bob}). \\ & \text{hay_fever}(\text{bob}). \end{aligned}$$

As can be seen from the example, LPADs encode in a natural way programs representing causal mechanisms: flu and hay fever are causes for sneezing, which, however, is probabilistic, in the sense that it may or may not happen even when the causes are present. The relationship between the DS, and LPADs in particular, and causal reasoning is discussed in Section 2.8.

2.1.2 ProbLog

The design of ProbLog [De Raedt et al., 2007] was motivated by the desire to make the simplest probabilistic extension of Prolog. In ProbLog, alternatives are expressed by *probabilistic facts* of the form

$$\Pi_i :: f_i$$

where $\Pi_i \in [0, 1]$ and f_i is an atom, meaning that each ground instantiation $f_i\theta$ of f_i is true with probability Π_i and false with probability $1 - \Pi_i$. Each world is obtained by selecting or rejecting each grounding of all probabilistic facts.

Example 13 (Medical symptoms – ProbLog). *Example 12 can be expressed in ProbLog as:*

```
sneezing(X) ← flu(X), flu_sneezing(X).
sneezing(X) ← hay_fever(X), hay_fever_sneezing(X).
flu(bob).
hay_fever(bob).
0.7 :: flu_sneezing(X).
0.8 :: hay_fever_sneezing(X).
```

2.1.3 Probabilistic Horn Abduction

Probabilistic Horn Abduction (PHA) [Poole, 1993b] and Independent Choice Logic (ICL) [Poole, 1997] express alternatives by facts, called *disjoint statements*, having the form

$$disjoint([a_{i1} : \Pi_{i1}, \dots, a_{in} : \Pi_{in}]).$$

where each a_{ik} is a logical atom and each Π_{ik} a number in $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} = 1$. Such a statement can be interpreted in terms of its ground instantiations: for each substitution θ grounding the atoms of the statement, the $a_{ik}\theta$ s are random alternatives and $a_{ik}\theta$ is true with probability Π_{ik} . Each world is obtained by selecting one atom from each grounding of each disjoint statement in the program. In practice, each ground instantiation of a disjoint statement corresponds to a random variable with as many values as the alternatives in the statement.

Example 14 (Medical symptoms – ICL). *Example 12 can be expressed in ICL as:*

$$\begin{aligned} sneezing(X) &\leftarrow flu(X), flu_sneezing(X). \\ sneezing(X) &\leftarrow hay_fever(X), hay_fever_sneezing(X). \\ flu(bob). \\ hay_fever(bob). \end{aligned}$$

$$\begin{aligned} disjoint([flu_sneezing(X) : 0.7, null : 0.3]). \\ disjoint([hay_fever_sneezing(X) : 0.8, null : 0.2]). \end{aligned}$$

In ICL, LPADs, and ProbLog, each grounding of a probabilistic clause is associated with a random variable with as many values as alternatives/head disjuncts for ICL and LPADs and with two values for ProbLog. The random variables corresponding to different instantiations of a probabilistic clause are independent and identically distributed (IID).

2.1.4 PRISM

The language PRISM [Sato and Kameya, 1997] is similar to PHA/ICL but introduces random facts via the predicate *msw/3* (*multi-switch*):

$$msw(SwitchName, TrialId, Value).$$

The first argument of this predicate is a *random switch name*, a term representing a set of discrete random variables; the second argument is an integer, the *trial id*; and the third argument represents a value for that variable. The set of possible values for a switch is defined by a fact of the form

$$values(SwitchName, [v_1, \dots, v_n]).$$

where *SwitchName* is again a term representing a switch name and each v_i is a term. Each ground pair (*SwitchName*, *TrialId*) represents a distinct random variable and the set of random variables associated with the same switch are IID.

The probability distribution over the values of the random variables associated with *SwitchName* is defined by a directive of the form

$$\leftarrow set_sw(SwitchName, [\Pi_1, \dots, \Pi_n]).$$

where p_i is the probability that variable *SwitchName* takes value v_i . Each world is obtained by selecting one value for each trial id of each random switch.

Example 15 (Coin tosses – PRISM). *The modeling of coin tosses shows differences in how the various PLP languages represent IID random variables. Suppose that coin c_1 is known not to be fair, but that all tosses of c_1 have the same probabilities of outcomes – in other words, each toss of c_1 is taken from a family of IID random variables. This can be represented in PRISM as*

$$\begin{aligned} & \text{values}(c_1, [\text{head}, \text{tail}]). \\ & \leftarrow \text{set_sw}(c_1, [0.4, 0.6]) \end{aligned}$$

Different tosses of c_1 can then be identified using the trial id argument of $msw/3$.

In PHA/ICL and many other PLP languages, each ground instantiation of a disjoint/1 statement represents a distinct random variable, so that IID random variables need to be represented through the statement’s instantiation patterns: e.g.,

$$\begin{aligned} & \text{disjoint}([\text{coin}(c_1, \text{TossNumber}, \text{head}) : 0.4, \\ & \text{coin}(c_1, \text{TossNumber}, \text{tail}) : 0.6]). \end{aligned}$$

In practice, the PRISM system accepts an $msw/2$ predicate whose atoms do not contain the trial id and for which each occurrence in a program is considered as being associated with a different new variable.

Example 16 (Medical symptoms – PRISM). *Example 14 can be encoded in PRISM as:*

$$\begin{aligned} & \text{sneezing}(X) \leftarrow \text{flu}(X), \text{msw}(\text{flu_sneezing}(X), 1). \\ & \text{sneezing}(X) \leftarrow \text{hay_fever}(X), \text{msw}(\text{hay_fever_sneezing}(X), 1). \\ & \text{flu}(\text{bob}). \\ & \text{hay_fever}(\text{bob}). \end{aligned}$$

$$\begin{aligned} & \text{values}(\text{flu_sneezing}(_X), [1, 0]). \\ & \text{values}(\text{hay_fever_sneezing}(_X), [1, 0]). \\ & \leftarrow \text{set_sw}(\text{flu_sneezing}(_X), [0.7, 0.3]). \\ & \leftarrow \text{set_sw}(\text{hay_fever_sneezing}(_X), [0.8, 0.2]). \end{aligned}$$

2.2 The Distribution Semantics for Programs Without Function Symbols

We present first the DS for the case of ProbLog as it is the language with the simplest syntax. A ProbLog program \mathcal{P} is composed by a set of normal

rules \mathcal{R} and a set \mathcal{F} of probabilistic facts. Each *probabilistic fact* is of the form $\Pi_i :: f_i$ where $\Pi_i \in [0, 1]$ and f_i is an atom¹, meaning that each ground instantiation $f_i\theta$ of f_i is true with probability Π_i and false with probability $1 - \Pi_i$. Each world is obtained by selecting or rejecting each grounding of each probabilistic fact.

An *atomic choice* indicates whether grounding $f\theta$ of a probabilistic fact $F = p :: f$ is selected or not. It is represented with the triple (f, θ, k) where $k \in \{0, 1\}$ and $k = 1$ means that the fact is selected, $k = 0$ that it is not. A set κ of atomic choices is *consistent* if it does not contain two atomic choices (f, θ, k) and (f, θ, j) with $k \neq j$ (only one alternative is selected for a ground probabilistic fact). The function *consistent*(κ) returns true if κ is consistent. A *composite choice* κ is a consistent set of atomic choices. The probability of composite choice κ is

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

A *selection* σ is a total composite choice, i.e., contains one atomic choice for every grounding of every probabilistic fact. A *world* w_σ is a logic program that is identified by a selection σ . The world w_σ is formed by including the atom corresponding to each atomic choice $(f, \theta, 1)$ of σ .

The probability of a world w_σ is $P(w_\sigma) = P(\sigma)$. Since in this section we are assuming programs without function symbols, the set of groundings of each probabilistic fact is finite, and so is the set of worlds $W_{\mathcal{P}}$. Accordingly, for a ProbLog program \mathcal{P} , $W_{\mathcal{P}} = \{w_1, \dots, w_m\}$. Moreover, $P(w)$ is a distribution over worlds: $\sum_{w \in W_{\mathcal{P}}} P(w) = 1$. We call *sound* a program for which every world has a two-valued WFM. We consider here sound programs, for non-sound ones, see Section 2.9.

Let q be a query in the form of a ground atom. We define the conditional probability of q given a world w as: $P(q|w) = 1$ if q is true in w and 0 otherwise. Since the program is sound, q can be only true or false in a world. The probability of q can thus be computed by summing out the worlds from the joint distribution of the query and the worlds:

$$P(q) = \sum_w P(q, w) = \sum_w P(q|w)P(w) = \sum_{w \models q} P(w). \quad (2.1)$$

This formula can also be used for computing the probability of a conjunction q_1, \dots, q_n of ground atoms since the truth of a conjunction of ground atoms

¹With an abuse of notation, sometimes we use \mathcal{F} to indicate the set containing the atoms f_i s. The meaning of \mathcal{F} will be clear from the context.

in a world is well defined. So we can compute the conditional probability of a query q given evidence e in the form of a conjunction of ground atoms e_1, \dots, e_m as

$$P(q|e) = \frac{P(q, e)}{P(e)} \quad (2.2)$$

We can also assign a probability to a query q by defining a probability space. Since $W_{\mathcal{P}}$ is finite, then $(W_{\mathcal{P}}, \mathbb{P}(W_{\mathcal{P}}))$ is a measurable space. For an element $\omega \in \mathbb{P}(W_{\mathcal{P}})$, define $\mu(\omega)$ as

$$\mu(\omega) = \sum_{w \in \omega} P(w)$$

with the probability of a world $P(w)$ defined as above. Then it's easy to see that $(W_{\mathcal{P}}, \mathbb{P}(W_{\mathcal{P}}), \mu)$ is a finitely additive probability space.

Given a ground atom q , define the function $Q : W_{\mathcal{P}} \rightarrow \{0, 1\}$ as

$$Q(w) = \begin{cases} 1 & \text{if } w \models q \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

Since the set of events is the powerset, then $Q^{-1}(\gamma) \in \mathbb{P}(W_{\mathcal{P}})$ for all $\gamma \subseteq \{0, 1\}$ and Q is a random variable. The distribution of Q is defined by $P(Q = 1)$ ($P(Q = 0)$ is given by $1 - P(Q = 1)$) and we indicate $P(Q = 1)$ with $P(q)$.

We can now compute $P(q)$ as

$$P(q) = \mu(Q^{-1}(\{1\})) = \mu(\{w | w \in W_{\mathcal{P}}, w \models q\}) = \sum_{w \models q} P(w)$$

obtaining the same formula as Equation (2.1).

The distribution over worlds also induces a distribution over interpretations: given an interpretation I , we can define the conditional probability of I given a world w as: $P(I|w) = 1$ if I is the model of w ($I \models w$) and 0 otherwise. The distribution over interpretations is then given by a formula similar to Equation (2.1):

$$P(I) = \sum_w P(I, w) = \sum_w P(I|w)P(w) = \sum_{I \models w} P(w) \quad (2.4)$$

We call the interpretations I for which $P(I) > 0$ *possible models* because they are models for at least one world.

Now define the function $\mathbf{I} : W_{\mathcal{P}} \rightarrow \{0, 1\}$ as

$$\mathbf{I}(I) = \begin{cases} 1 & \text{if } I \models w \\ 0 & \text{otherwise} \end{cases} \quad (2.5)$$

$\mathbf{I}^{-1}(\gamma) \in \mathbb{P}(W_{\mathcal{P}})$ for all $\gamma \subseteq \{0, 1\}$ so \mathbf{I} is a random variable for probability space $(W_{\mathcal{P}}, \mathbb{P}(W_{\mathcal{P}}), \mu)$. The distribution of \mathbf{I} is defined by $P(\mathbf{I} = 1)$ and we indicate $P(\mathbf{I} = 1)$ with $P(I)$.

We can now compute $P(I)$ as

$$P(I) = \mu(\mathbf{I}^{-1}(\{1\})) = \mu(\{w \mid w \in W_{\mathcal{P}}, I \models w\}) = \sum_{I \models w} P(w)$$

obtaining the same formula as Equation (2.4).

The probability of a query q can be obtained from the distribution over interpretations by defining the conditional probability of q given an interpretation I as $P(q|I) = 1$ if $I \models q$ and 0 otherwise and by marginalizing the interpretations obtaining

$$P(q) = \sum_I P(q, I) = \sum_I P(q|I)P(I) = \sum_{I \models q} P(I) = \sum_{I \models q, I \models w} P(w) \quad (2.6)$$

So the probability of a query can be obtained by summing the probability of the possible models where the query is true.

Example 17 (Medical symptoms – worlds – ProbLog). *Consider the program of Example 13. The program has four worlds*

$$\begin{array}{ll} w_1 = \{ & w_2 = \{ \\ & \quad flu_sneezing(bob). \\ & \quad hay_fever_sneezing(bob). \\ \} & \} \\ P(w_1) = & P(w_2) = \\ 0.7 \times 0.8 & 0.3 \times 0.8 \\ w_3 = \{ & w_4 = \{ \\ & \quad flu_sneezing(bob). \\ \} & \} \\ P(w_3) = & P(w_4) = \\ 0.7 \times 0.2 & 0.3 \times 0.2 \end{array}$$

The query *sneezing(bob)* is true in three worlds and its probability

$$P(\text{sneezing}(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94.$$

Note that the contributions from the two clauses are combined disjunctively. The probability of the query is thus computed using the rule giving the probability of the disjunction of two independent Boolean random variables:

$$P(a \vee b) = P(a) + P(b) - P(a)P(b) = 1 - (1 - P(a))(1 - P(b)).$$

In our case, $P(\text{sneezing}(\text{bob})) = 0.7 + 0.8 - 0.7 \cdot 0.8 = 0.94$.

We now give the semantics for LPADs. A clause

$$C_i = h_{i1} : \Pi_{i1} ; \dots ; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \dots, b_{im_i}$$

stands for a set of probabilistic clauses, one for each ground instantiation $C_i\theta$ of C_i . Each ground probabilistic clause represents a choice among n_i normal clauses, each of the form

$$h_{ik} \leftarrow b_{i1}, \dots, b_{im_i}$$

for $k = 1 \dots, n_i$. Moreover, another clause

$$\text{null} \leftarrow b_{i1}, \dots, b_{im_i}$$

is implicitly encoded which is associated with probability $\Pi_0 = 1 - \sum_{k=1}^{n_i} \Pi_k$. So for LPAD P an *atomic choice* is the selection of a head atom for a grounding $C_i\theta_j$ of a probabilistic clause C_i , including the atom *null*. An atomic choice is represented in this case by the triple (C_i, θ_j, k) , where θ_j is a grounding substitution and $k \in \{0, 1, \dots, n_i\}$. An atomic choice represents an equation of the form $X_{ij} = k$ where X_{ij} is a random variable associated with $C_i\theta_j$. The definition of consistent set of atomic choices, of composite choices, and of the probability of a composite choice is the same as for ProbLog. Again, a *selection* σ is a total composite choice (one atomic choice for every grounding of each probabilistic clause). A selection σ identifies a logic program w_σ (a *world*) that contains the normal clauses obtained by selecting head atom $h_{ik}\theta$ for each atomic choice (C_i, θ, k) :

$$w_\sigma = \{ (h_{ik} \leftarrow b_{i1}, \dots, b_{im_i})\theta \mid (C_i, \theta, k) \in \sigma, \\ C_i = h_{i1} : \Pi_{i1} ; \dots ; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \dots, b_{im_i}, C_i \in \mathcal{P} \}$$

As for ProbLog, the probability of w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(C_i, \theta_j, k) \in \sigma} \Pi_{ik}$, the set of worlds $W_P = \{w_1, \dots, w_m\}$ is finite, and $P(w)$ is a distribution over worlds.

If q is a query, we can define $P(q|w)$ as for ProbLog and again the probability of q is given by Equation (2.1)

Example 18 (Medical symptoms – worlds – LPAD). *The LPAD of Example 12 has four worlds:*

$$\begin{aligned}
 w_1 &= \{ \\
 &\quad sneezing(bob) \leftarrow flu(bob). \\
 &\quad sneezing(bob) \leftarrow hay_fever(bob). \\
 &\quad flu(bob). \quad hay_fever(bob). \\
 &\quad \} \\
 P(w_1) &= 0.7 \times 0.8 \\
 w_2 &= \{ \\
 &\quad null \leftarrow flu(bob). \\
 &\quad sneezing(bob) \leftarrow hay_fever(bob). \\
 &\quad flu(bob). \quad hay_fever(bob). \\
 &\quad \} \\
 P(w_2) &= 0.3 \times 0.8 \\
 w_3 &= \{ \\
 &\quad sneezing(bob) \leftarrow flu(bob). \\
 &\quad null \leftarrow hay_fever(bob). \\
 &\quad flu(bob). \quad hay_fever(bob). \\
 &\quad \} \\
 P(w_3) &= 0.7 \times 0.2 \\
 w_4 &= \{ \\
 &\quad null \leftarrow flu(bob). \\
 &\quad null \leftarrow hay_fever(bob). \\
 &\quad flu(bob). \quad hay_fever(bob). \\
 &\quad \} \\
 P(w_4) &= 0.3 \times 0.2
 \end{aligned}$$

sneezing(bob) is true in three worlds and its probability is

$$P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$$

2.3 Examples of Programs

In this section, we provide some examples of programs to better illustrate the syntax and the semantics.

Example 19 (Detailed medical symptoms – LPAD). *The following LPAD² models a program that describe medical symptoms in a way that is slightly more elaborated than Example 12:*

```

strong_sneezing(X) : 0.3 ; moderate_sneezing(X) : 0.5 ←
    flu(X).
strong_sneezing(X) : 0.2 ; moderate_sneezing(X) : 0.6 ←
    hay_fever(X).
flu(bob).
hay_fever(bob).

```

Here the clauses have three alternatives in the head of which the one associated with atom null is left implicit. This program has nine worlds, the query `strong_sneezing(bob)` is true in five of them, and $P(\text{strong_sneezing}(\text{bob})) = 0.44$.

Example 20 (Coin – LPAD). *The coin example of [Vennekens et al., 2004] is represented as³:*

```

heads(Coin) : 1/2 ; tails(Coin) : 1/2 ←
    toss(Coin), ~biased(Coin).
heads(Coin) : 0.6 ; tails(Coin) : 0.4 ←
    toss(Coin), biased(Coin).
fair(Coin) : 0.9 ; biased(Coin) : 0.1.
toss(coin).

```

The first clause states that, if we toss a coin that is not biased, it has equal probability of landing heads and tails. The second states that, if the coin is biased, it has a slightly higher probability of landing heads. The third states that the coin is fair with probability 0.9 and biased with probability 0.1 and the last clause states that we toss the coin with certainty. This program has eight worlds, the query `heads(coin)` is true in four of them, and its probability is 0.51.

Example 21 (Eruption – LPAD). *Consider this LPAD⁴ from Riguzzi and Di Mauro [2012] that is inspired by the morphological characteristics of the Italian island of Stromboli:*

²<http://cplint.eu/e/sneezing.pl>

³<http://cplint.eu/e/coin.pl>

⁴<http://cplint.eu/e/eruption.pl>

$C_1 = \text{eruption} : 0.6 ; \text{earthquake} : 0.3 :- \text{sudden_energy_release},$
 $\text{fault_rupture}(X).$
 $C_2 = \text{sudden_energy_release} : 0.7.$
 $C_3 = \text{fault_rupture}(\text{southwest_northeast}).$
 $C_4 = \text{fault_rupture}(\text{east_west}).$

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 models the possibility that an eruption or an earthquake occurs at Stromboli. If there is a sudden energy release under the island and there is a fault rupture, then 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 in both faults.

Clause C_1 has two groundings, $C_1\theta_1$ with

$$\theta_1 = \{X/\text{southwest_northeast}\}$$

and $C_1\theta_2$ with

$$\theta_2 = \{X/\text{east_west}\},$$

while clause C_2 has a single grounding $C_2\emptyset$. Since C_1 has three head atoms and C_2 two, the program has $3 \times 3 \times 2$ worlds. The query eruption is true in five of them and its probability is $P(\text{eruption}) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$.

Example 22 (Monty Hall puzzle – LPAD). The Monty Hall puzzle [Baral et al., 2009] refers to the TV game show hosted by Monty Hall in which a player has to choose which of three closed doors to open. Behind one door, there is a prize, while behind the other two, there is nothing. Once the player has selected the door, Monty Hall opens one of the remaining closed doors which does not contain the prize, and then he asks the player if he would like to change his door with the other closed door or not. The problem of this game is to determine whether the player should switch. The following program provides a solution⁵. The prize is behind one of the three doors with the same probability:

$\text{prize}(1) : 1/3 ; \text{prize}(2) : 1/3 ; \text{prize}(3) : 1/3.$

The player has selected door 1:

$\text{selected}(1).$

⁵<http://cplint.eu/e/monty.swinb>

Monty opens door 2 with probability 0.5 and door 3 with probability 0.5 if the prize is behind door 1:

$open_door(2) : 0.5 ; open_door(3) : 0.5 \leftarrow prize(1).$

Monty opens door 2 if the prize is behind door 3:

$open_door(2) \leftarrow prize(3).$

Monty opens door 3 if the prize is behind door 2:

$open_door(3) \leftarrow prize(2).$

The player keeps his choice and wins if he has selected a door with the prize:

$win_keep \leftarrow prize(1).$

The player switches and wins if the prize is behind the door that he has not selected and that Monty did not open:

$win_switch \leftarrow prize(2), open_door(3).$

$win_switch \leftarrow prize(3), open_door(2).$

Querying win_keep and win_switch we obtain probability $1/3$ and $2/3$ respectively, so the player should switch. Note that if you change the probability distribution of Monty selecting a door to open when the prize is behind the door selected by the player, then the probability of winning by switching remains the same.

Example 23 (Three-prisoner puzzle – LPAD). The following program⁶ from [Riguzzi et al., 2016a] encodes the three-prisoner puzzle. In Grünwald and Halpern [2003], the problem is described as:

Of three prisoners a, b, and c, two are to be executed, but a does not know which. Thus, a thinks that the probability that i will be executed is $2/3$ for $i \in \{a, b, c\}$. He says to the jailer, “Since either b or c is certainly going to be executed, you will give me no information about my own chances if you give me the name of one man, either b or c, who is going to be executed.” But then, no matter what the jailer says, naive conditioning leads a to believe that his chance of execution went down from $2/3$ to $1/2$.

Each prisoner is safe with probability $1/3$:

$safe(a) : 1/3 ; safe(b) : 1/3 ; safe(c) : 1/3.$

If a is safe, the jailer tells that one of the other prisoners will be executed uniformly at random:

$tell_executed(b) : 1/2 ; tell_executed(c) : 1/2 \leftarrow safe(a).$

Otherwise, he tells that the only unsafe prisoner will be executed:

⁶<http://cplint.eu/ef/jail.swinb>

$tell_executed(b) \leftarrow safe(c).$

$tell_executed(c) \leftarrow safe(b).$

The jailer speaks if he tells that somebody will be executed:

$tell \leftarrow tell_executed(_).$

a is safe after the jailer utterance if he is safe and the jailer speaks:

$safe_after_tell : -safe(a), tell.$

By computing the probability of $safe(a)$ and $safe_after_tell$, we get the same probability of $1/3$, so the jailer utterance does not change the probability of a of being safe.

We can see this also by considering conditional probabilities: the probability of $safe(a)$ given the jailer utterance $tell$ is

$$P(safe(a)|tell) = \frac{P(safe(a), tell)}{P(tell)} = \frac{P(safe_after_tell)}{P(tell)} = \frac{1/3}{1} = 1/3$$

because the probability of $tell$ is 1.

Example 24 (Russian roulette with two guns – LPAD). The following example⁷ models a Russian roulette game with two guns [Baral et al., 2009]. The death of the player is caused with probability $1/6$ by triggering the left gun and similarly for the right gun:

$death : 1/6 \leftarrow pull_trigger(left_gun).$

$death : 1/6 \leftarrow pull_trigger(right_gun).$

$pull_trigger(left_gun).$

$pull_trigger(right_gun).$

Querying the probability of death we get the probability of the player of dying.

Example 25 (Mendelian rules of inheritance – LPAD). Blockeel [2004] presents a program⁸ that encodes the Mendelian rules of inheritance of the color of pea plants. The color of a pea plant is determined by a gene that exists in two forms (alleles), purple, p , and white, w . Each plant has two alleles for the color gene that reside on a couple of chromosomes. $cg(X, N, A)$ indicates that plant X has allele A on chromosome N . The program is:

$color(X, white) \leftarrow cg(X, 1, w), cg(X, 2, w).$

$color(X, purple) \leftarrow cg(X, _A, p).$

⁷<http://cplint.eu/e/trigger.pl>

⁸<http://cplint.eu/e/mendel.pl>

```

cg(X, 1, A) : 0.5 ; cg(X, 1, B) : 0.5 ←
  mother(Y, X), cg(Y, 1, A), cg(Y, 2, B).
cg(X, 2, A) : 0.5 ; cg(X, 2, B) : 0.5 ←
  father(Y, X), cg(Y, 1, A), cg(Y, 2, B).
mother(m, c).  father(f, c).
cg(m, 1, w).  cg(m, 2, w).  cg(f, 1, p).  cg(f, 2, w).

```

The facts of the program express that *c* is the offspring of *m* and *f* and that the alleles of *m* are *ww* and of *f* are *pw*. The disjunctive rules encode the fact that an offspring inherits the allele on chromosome 1 from the mother and the allele on chromosome 2 from the father. In particular, each allele of the parent has a probability of 50% of being transmitted. The definite clauses for color express the fact that the color of a plant is purple if at least one of the alleles is *p*, i.e., that the *p* allele is dominant. In a similar way, the rules of blood type inheritance can be written in an LPAD⁹.

Example 26 (Path probability – LPAD). An interesting application of PLP under the DS is the computation of the probability of a path between two nodes in a graph in which the presence of each edge is probabilistic¹⁰:

```

path(X, X).
path(X, Y) ← path(X, Z), edge(Z, Y).
edge(a, b) : 0.3.  edge(b, c) : 0.2.  edge(a, c) : 0.6.

```

This program, coded in ProbLog, was used in [De Raedt et al., 2007] for computing the probability that two biological concepts are related in the BIOMINE network [Sevon et al., 2006].

PLP under the DS can encode BNs Vennekens et al. [2004]: each value of each random variable is encoded by a ground atom, each row of each CPT is encoded by a rule with the value of parents in the body and the probability distribution of values of the child in the head.

Example 27 (Alarm BN – LPAD). For example, the BN of Example 10 that we repeat in Figure 2.1 for readability can be encoded with the program¹¹

⁹<http://cplint.eu/e/bloodtype.pl>

¹⁰<http://cplint.eu/e/path.swinb>

¹¹<http://cplint.eu/e/alarm.pl>

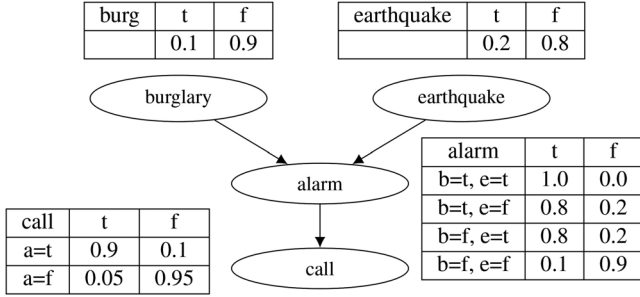


Figure 2.1 Example of a BN.

$burg(t) : 0.1 ; burg(f) : 0.9.$
 $earthquake(t) : 0.2 ; earthquake(f) : 0.8.$
 $alarm(t) \leftarrow burg(t), earthq(t).$
 $alarm(t) : 0.8 ; alarm(f) : 0.2 \leftarrow burg(t), earthq(f).$
 $alarm(t) : 0.8 ; alarm(f) : 0.2 \leftarrow burg(f), earthq(t).$
 $alarm(t) : 0.1 ; alarm(f) : 0.9 \leftarrow burg(f), earthq(f).$
 $call(t) : 0.9 ; call(f) : 0.1 \leftarrow alarm(t).$
 $call(t) : 0.05 ; call(f) : 0.95 \leftarrow alarm(f).$

2.4 Equivalence of Expressive Power

To show that all these languages have the same expressive power, we discuss transformations among probabilistic constructs from the various languages.

The mapping between PHA/ICL and PRISM translates each PHA/ICL disjoint statement to a multi-switch declaration and vice versa in the obvious way. The mapping from PHA/ICL and PRISM to LPADs translates each disjoint statement/multi-switch declaration to a disjunctive LPAD fact.

The translation from an LPAD into PHA/ICL (first shown in [Vennekens and Verbaeten, 2003]) rewrites each clause C_i with v variables \bar{X}

$$h_1 : \Pi_1 ; \dots ; h_n : \Pi_n \leftarrow B.$$

into PHA/ICL by adding n new predicates $\{choice_{i1}/v, \dots, choice_{in}/v\}$ and a disjoint statement:

$$h_1 \leftarrow B, \text{choice}_{i_1}(\bar{X}).$$

$$\vdots$$

$$h_n \leftarrow B, \text{choice}_{i_n}(\bar{X}).$$

$$\text{disjoint}([\text{choice}_{i_1}(\bar{X}) : \Pi_1, \dots, \text{choice}_{i_n}(\bar{X}) : \Pi_n]).$$

For instance, the first clause of the medical symptoms LPAD of Example 19 is translated to

$$\text{strong_sneezing}(X) \leftarrow \text{flu}(X), \text{choice}_{11}(X).$$

$$\text{moderate_sneezing}(X) : 0.5 \leftarrow \text{flu}(X), \text{choice}_{12}(X).$$

$$\text{disjoint}([\text{choice}_{11}(X) : 0.3, \text{choice}_{12}(X) : 0.5, \text{choice}_{13} : 0.2]).$$

where the clause $\text{null} \leftarrow \text{flu}(X), \text{choice}_{13}$. is omitted since null does not appear in the body of any clause.

Finally, as shown in [De Raedt et al., 2008], to convert LPADs into ProbLog, each clause C_i with v variables \bar{X}

$$h_1 : \Pi_1 ; \dots ; h_n : \Pi_n \leftarrow B.$$

is translated into ProbLog by adding $n - 1$ probabilistic facts for predicates $\{f_{i_1/v}, \dots, f_{i_n/v}\}$:

$$h_1 \leftarrow B, f_{i_1}(\bar{X}).$$

$$h_2 \leftarrow B, \sim f_{i_1}(\bar{X}), f_{i_2}(\bar{X}).$$

$$\vdots$$

$$h_n \leftarrow B, \sim f_{i_1}(\bar{X}), \dots, \sim f_{i_{n-1}}(\bar{X}).$$

$$\pi_1 :: f_{i_1}(\bar{X}).$$

$$\vdots$$

$$\pi_{n-1} :: f_{i_{n-1}}(\bar{X}).$$

where

$$\pi_1 = \Pi_1$$

$$\pi_2 = \frac{\Pi_2}{1 - \pi_1}$$

$$\pi_3 = \frac{\Pi_3}{(1 - \pi_1)(1 - \pi_2)}$$

$$\dots$$

In general

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

Note that while the translation into ProbLog introduces negation, the introduced negation involves only probabilistic facts, and so the transformed program will have a two-valued model whenever the original program does.

For instance, the first clause of the medical symptoms LPAD of Example 19 is translated to

$$\begin{aligned} \text{strong_sneezing}(X) &\leftarrow \text{flu}(X), f_{11}(X). \\ \text{moderate_sneezing}(X) : 0.5 &\leftarrow \text{flu}(X), \sim f_{11}(X), f_{12}(X). \\ 0.3 &:: f_{11}(X). \\ 0.71428571428 &:: f_{12}(X). \end{aligned}$$

2.5 Translation to Bayesian Networks

We discuss here how an acyclic ground LPAD can be translated to a BN. Let us first define the acyclic property for LPADs, extending Definition 4. An LPAD is *acyclic* if an integer level can be assigned to each ground atom so that the level of each atom in the head of each ground rule is the same and is higher than the level of each atom in the body.

An acyclic ground LPAD \mathcal{P} can be translated to a BN $\beta(\mathcal{P})$ [Vennekens et al., 2004]. $\beta(\mathcal{P})$ is built by associating each atom a in $\mathcal{B}_{\mathcal{P}}$ with a binary variable a with values true (1) and false (0). Moreover, for each rule C_i of the following form

$$h_1 : \Pi_1 ; \dots ; h_n : \Pi_n \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_l$$

in $\text{ground}(\mathcal{P})$, we add a new variable ch_i (for “choice for rule C_i ”) to $\beta(\mathcal{P})$. ch_i has $b_1, \dots, b_m, c_1, \dots, c_l$ as parents. The values for ch_i are h_1, \dots, h_n and *null*, corresponding to the head atoms. The CPT of ch_i is

	...	$b_1 = 1, \dots, b_m = 1, c_1 = 0, \dots, c_l = 0$...
$\text{ch}_i = h_1$	0.0	Π_1	0.0
...			
$\text{ch}_i = h_n$	0.0	Π_n	0.0
$\text{ch}_i = \text{null}$	1.0	$1 - \sum_{i=1}^n \Pi_i$	1.0

that can be expressed as

$$P(\text{ch}_i | b_1, \dots, c_l) = \begin{cases} \Pi_k & \text{if } \text{ch}_i = h_k, b_i = 1, \dots, c_l = 0 \\ 1 - \sum_{j=1}^n \Pi_j & \text{if } \text{ch}_i = \text{null}, b_i = 1, \dots, c_l = 0 \\ 1 & \text{if } \text{ch}_i = \text{null}, \neg(b_i = 1, \dots, c_l = 0) \\ 0 & \text{otherwise} \end{cases} \quad (2.7)$$

If the body is empty, the CPT for ch_i is

$ch_i = h_1$	Π_1
\dots	
$ch_n = h_n$	Π_n
$ch_i = null$	$1 - \sum_{i=1}^n \Pi_i$

Moreover, for each variable a corresponding to atom $a \in \mathcal{B}_P$, the parents are all the variables ch_i of rules C_i that have a in the head. The CPT for a is the following deterministic table:

	At least one parent equal to a	Remaining columns
$a = 1$	1.0	0.0
$a = 0$	0.0	1.0

encoding the function

$$a = f(\mathbf{ch}_a) = \begin{cases} 1 & \text{if } \exists ch_i \in \mathbf{ch}_a : ch_i = a \\ 0 & \text{otherwise} \end{cases}$$

where \mathbf{ch}_a are the parents of a . Note that in order to convert an LPAD containing variables into a BN, its grounding must be generated.

Example 28 (LPAD to BN). *Consider the following LPAD \mathcal{P} :*

- $C_1 = a_1 : 0.4 ; a_2 : 0.3.$
- $C_2 = a_2 : 0.1 ; a_3 : 0.2.$
- $C_3 = a_4 : 0.6 ; a_5 : 0.4 \leftarrow a_1.$
- $C_4 = a_5 : 0.4 \leftarrow a_2, a_3.$
- $C_5 = a_6 : 0.3 ; a_7 : 0.2 \leftarrow a_2, a_5.$

Its corresponding network $\beta(\mathcal{P})$ is shown in Figure 1.7, where the CPT for a_2 and ch_5 are shown in Tables 2.1 and 2.2 respectively.

Table 2.1 Conditional probability table for a_2

ch_1, ch_2	a_1, a_2	a_1, a_3	a_2, a_2	a_2, a_3
$a_2 = 1$	1.0	0.0	1.0	1.0
$a_2 = 0$	0.0	1.0	0.0	0.0

Table 2.2 Conditional probability table for ch_5

a_2, a_5	1,1	1,0	0,1	0,0
$ch_5 = x_6$	0.3	0.0	0.0	0.0
$ch_5 = x_7$	0.2	0.0	0.0	0.0
$ch_5 = null$	0.5	1.0	1.0	1.0

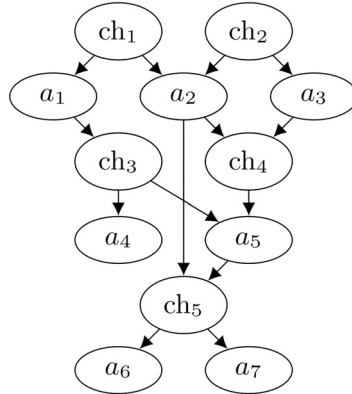


Figure 2.2 BN $\beta(\mathcal{P})$ equivalent to the program of Example 28.

An alternative translation $\gamma(\mathcal{P})$ for a ground program P is built by including random variables a for each atom a in $\mathcal{B}_{\mathcal{P}}$ and ch_i for each clause C_i as for $\beta(\mathcal{P})$. Moreover, $\gamma(\mathcal{P})$ includes the Boolean random variable $body_i$ and the random variable X_i with values h_1, \dots, h_n and $null$ for each clause C_i .

The parents of $body_i$ are b_1, \dots, b_m , and c_1, \dots, c_l and its CPT encodes the deterministic AND Boolean function:

	...	$b_1 = 1, \dots, b_m = 1, c_1 = 0, \dots, c_l = 0$...
$body_i = 0$	1.0	0.0	1.0
$body_i = 1$	0.0	1.0	0.0

If the body is empty, the CPT makes $body_i$ surely true

$body_i = 0$	0.0
$body_i = 1$	1.0

X_i has no parents and has the CPT

$ch_i = h_1$	Π_1
...	
$ch_i = h_n$	Π_n
$ch_i = null$	$1 - \sum_{i=1}^n \Pi_i$

ch_i has X_i and $body_i$ as parents with the deterministic CPT

$body_i, X_i$	$0, h_1$...	$0, h_n$	$0, null$	$1, h_1$...	$1, h_n$	$1, null$
$ch_i = h_1$	0.0	...	0.0	0.0	1.0	...	0.0	0.0
...								
$ch_i = h_n$	0.0	...	0.0	0.0	0.0	...	1.0	0.0
$ch_i = null$	1.0	...	1.0	1.0	0.0	...	0.0	1.0

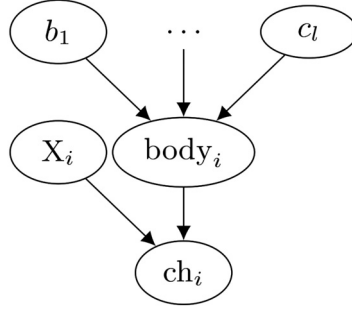


Figure 2.3 Portion of $\gamma(\mathcal{P})$ relative to a clause C_i .

encoding the function

$$ch_i = f(body_i, X_i) = \begin{cases} X_i & \text{if } body_i = 1 \\ null & \text{if } body_i = 0 \end{cases}$$

The parents of each variable a in $\gamma(\mathcal{P})$ are the variables ch_i of rules C_i that have a in the head as for $\beta(\mathcal{P})$, with the same CPT as in $\beta(\mathcal{P})$.

The portion of $\gamma(\mathcal{P})$ relative to a clause C_i is shown in Figure 2.3.

If we compute $P(ch_i|b_1, \dots, b_m, c_1, \dots, c_l)$ by marginalizing

$$P(ch_i, body_i, X_i|b_1, \dots, b_m, c_1, \dots, c_l)$$

we can see that we obtain the same dependency as in $\beta(\mathcal{P})$:

$$\begin{aligned} P(ch_i|b_1, \dots, c_l) &= \\ &= \sum_{x_i} \sum_{body_i} P(ch_i, body_i, x_i|b_1, \dots, c_l) \\ &= \sum_{x_i} \sum_{body_i} P(ch_i|body_i, x_i)P(x_i)P(body_i|b_1, \dots, c_l) \\ &= \sum_{x_i} P(x_i) \sum_{body_i} P(ch_i|body_i, x_i)P(body_i|b_1, \dots, c_l) \\ &= \sum_{x_i} P(x_i) \sum_{body_i} P(ch_i|body_i, x_i) \begin{cases} 1 & \text{if } body_i = 1, b_1 = 1, \dots, c_l = 0 \\ 1 & \text{if } body_i = 0, \neg(b_1 = 1, \dots, c_l = 0) \\ 0 & \text{otherwise} \end{cases} \\ &= \sum_{x_i} P(x_i) \sum_{body_i} \begin{cases} 1 & \text{if } ch_i = x_i, body_i = 1, b_1 = 1, \dots, c_l = 0 \\ 1 & \text{if } ch_i = null, body_i = 0, \neg(b_1 = 1, \dots, c_l = 0) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

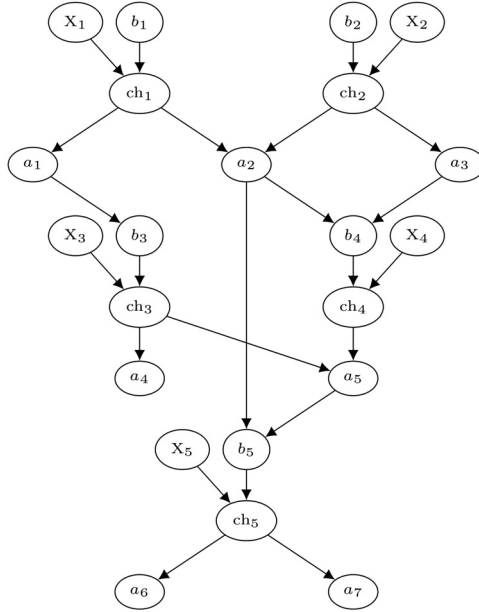


Figure 2.4 BN $\gamma(\mathcal{P})$ equivalent to the program of Example 28.

$$\begin{aligned}
 &= \sum_{x_i} P(x_i) \begin{cases} 1 & \text{if } ch_i = x_i, b_1 = 1, \dots, c_l = 0 \\ 1 & \text{if } ch_i = \text{null}, \neg(b_1 = 1, \dots, c_l = 0) \\ 0 & \text{otherwise} \end{cases} \\
 &= \begin{cases} \prod_k & \text{if } ch_i = h_k, b_i = 1, \dots, c_l = 0 \\ 1 - \sum_{j=1}^n \Pi_j & \text{if } ch_i = \text{null}, b_i = 1, \dots, c_l = 0 \\ 1 & \text{if } ch_i = \text{null}, \neg(b_i = 1, \dots, c_l = 0) \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

which is the same as Equation (2.7).

From Figure 2.3 and using d-separation (see Definition 17), we can see that the X_i variables are all pairwise unconditionally independent as between every couple there is the collider $X_i \rightarrow ch_i \leftarrow \text{body}_i$.

Figure 2.4 shows $\gamma(\mathcal{P})$ for Example 28.

2.6 Generality of the Distribution Semantics

The assumption of independence of the random variables associated with ground clauses may seem restrictive. However, any probabilistic relationship between Boolean random variables that can be represented with a BN can be

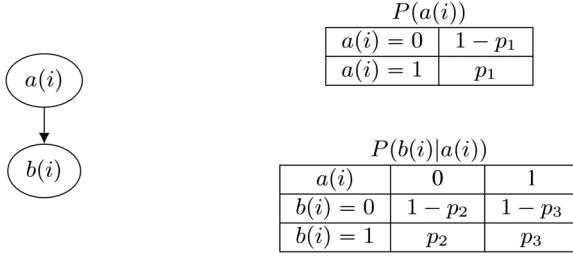


Figure 2.5 BN representing the dependency between $a(i)$ and $b(i)$.

modeled in this way. For example, suppose you want to model a general dependency between the ground atoms $a(i)$ and $b(i)$ regarding predicates $a/1$ and $b/1$ and constant i . This dependency can be represented with the BN of Figure 2.5.

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

$$\begin{aligned}
 P(0, 0) &= (1 - p_1)(1 - p_2) \\
 P(0, 1) &= (1 - p_1)p_2 \\
 P(1, 0) &= p_1(1 - p_3) \\
 P(1, 1) &= p_1p_3
 \end{aligned}$$

This dependency can be modeled with the following LPAD \mathcal{P} :

$$\begin{aligned}
 C_1 &= a(i) : p_1 \\
 C_2 &= b(X) : p_2 \leftarrow a(X) \\
 C_3 &= b(X) : p_3 \leftarrow \sim a(X)
 \end{aligned}$$

We can associate Boolean random variables X_1 with C_1 , X_2 , with $C_2\{X/i\}$, and X_3 with $C_3\{X/i\}$, where X_1 , X_2 , and X_3 are mutually independent. These three random variables generate eight worlds. $\neg a(i) \wedge \neg b(i)$ for example is true in the worlds

$$w_1 = \emptyset, w_2 = \{b(i) \leftarrow a(i)\}$$

whose probabilities are

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

so

$$P'(\neg a(i), \neg b(i)) = (1 - p_1)(1 - p_2)(1 - p_3) + (1 - p_1)(1 - p_2)p_3 = P(0, 0).$$

We can prove similarly that the distributions P and P' coincide for all joint states of $a(i)$ and $b(i)$.

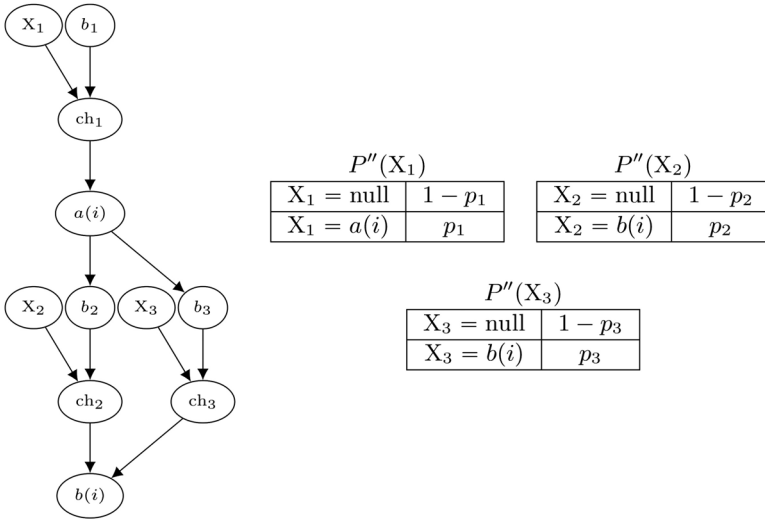


Figure 2.6 BN modeling the distribution over $a(i)$, $b(i)$, X_1 , X_2 , X_3 .

Modeling the dependency between $a(i)$ and $b(i)$ with the program above is equivalent to represent the BN of Figure 2.5 with the network $\gamma(\mathcal{P})$ of Figure 2.6.

Since $\gamma(\mathcal{P})$ defines the same distribution as \mathcal{P} , the distributions P and P'' , the one defined by $\gamma(\mathcal{P})$, agree on the variables $a(i)$ and $b(i)$, i.e.,

$$P(a(i), b(i)) = P''(a(i), b(i))$$

for any value of $a(i)$ and $b(i)$. From Figure 2.6, it is also clear that X_1 , X_2 , and X_3 are mutually unconditionally independent, thus showing that it is possible to represent any dependency with independent random variables. So we can model general dependencies among ground atoms with the DS.

This confirms the results of Sections 2.3 and 2.5 that graphical models can be translated into probabilistic logic programs under the DS and vice versa. Therefore, the two formalisms are equally expressive.

2.7 Extensions of the Distribution Semantics

Programs under the DS may contain *flexible probabilities* [De Raedt and Kimmig, 2015] or probabilities that depend on values computed during program execution. In this case, the probabilistic annotations are variables, as in the program¹² from [De Raedt and Kimmig, 2015]

¹²<http://cplint.eu/e/flexprob.pl>

```
red(Prob) : Prob.
```

```
draw_red(R, G) :-
  Prob is R / (R + G),
  red(Prob).
```

The query `draw_red(r, g)`, where `r` and `g` are the number of green and red balls in an urn, succeeds with the same probability as that of drawing a red ball from the urn.

Flexible probabilities allow the computation of probabilities on the fly during inference. However, flexible probabilities must be ground when their value must be evaluated during inference. Many inference systems support them by imposing constraints on the form of programs.

The body of rules may also contain literals for a meta-predicate such as `prob/2` that computes the probability of an atom, thus allowing nested or meta-probability computations [De Raedt and Kimmig, 2015]. Among the possible uses of such a feature De Raedt and Kimmig [2015] mention: filtering proofs on the basis of the probability of subqueries, or implementing simple forms of combining rules.

An example of the first use is¹³

```
a:0.2 :-
  prob(b, P),
  P > 0.1.
```

where `a` succeeds with probability 0.2 only if the probability of `b` is larger than 0.1.

An example of the latter is¹⁴

```
p(P) : P.

max_true(G1, G2) :-
  prob(G1, P1),
  prob(G2, P2),
  max(P1, P2, P), p(P).
```

where `max_true(G1, G2)` succeeds with the success probability of its more likely argument.

¹³<http://cplint.eu/e/meta.pl>

¹⁴<http://cplint.eu/e/metacomb.pl>

2.8 CP-Logic

CP-logic [Vennekens et al., 2009] is a language for representing causal laws. It shares many similarities with LPADs but specifically aims at modeling probabilistic causality. Syntactically, *CP-logic programs*, or *CP-theories*, are identical to lpads¹⁵: they are composed of annotated disjunctive clauses that are interpreted as follows: for each grounding

$$h_1 : \Pi_1 ; \dots ; h_m : \Pi_m \leftarrow B$$

of a clause of the program, B represents an event whose effect is to cause at most one of the h_i atoms to become true and the probability of h_i of being caused is Π_i . Consider the following medical example.

Example 29 (CP-logic program – infection [Vennekens et al., 2009]). *A patient is infected by a bacterium. Infection can cause either pneumonia or angina. In turn, angina can cause pneumonia and pneumonia can cause angina. This can be represented by the CP-logic program:*

$$\text{angina} : 0.2 \leftarrow \text{pneumonia}. \quad (2.8)$$

$$\text{pneumonia} : 0.3 \leftarrow \text{angina}. \quad (2.9)$$

$$\text{pneumonia} : 0.4 ; \text{angina} : 0.1 \leftarrow \text{infection}. \quad (2.10)$$

$$\text{infection}. \quad (2.11)$$

The semantics of CP-logic programs is given in terms of probability trees that represent the possible courses of the events encoded in the program. We consider first the case where the program is positive, i.e., the bodies of rules do not contain negative literals.

Definition 18 (Probability tree – positive case). *A probability tree¹⁶ T for a program \mathcal{P} is a tree where every node n is labeled with a two-valued interpretation $I(n)$ and a probability $P(n)$. T is constructed as follows:*

- *The root node r has probability $P(r) = 1.0$ and interpretation $I(r) = \emptyset$.*
- *Each inner node n is associated with a ground clause C_i such that*
 - *no ancestor of n is associated with C_i ,*
 - *all atoms in $\text{body}(C_i)$ are true in $I(n)$,*

¹⁵There are versions of CP-logic that have a more general syntax but they are not essential for the discussion here

¹⁶We follow here the definition of [Shterionov et al., 2015] for its simplicity.

n has one child node for each atom $h_k \in \text{head}(C_i)$. The k -th child has interpretation $I(n) \cup \{h_k\}$ and probability $P(n) \cdot \Pi_k$.

- No leaf can be associated with a clause following the rule above.

A probability tree defines a probability distribution $P(I)$ over the interpretation of the program \mathcal{P} : the probability of an interpretation I is the sum of the probabilities of the leaf nodes n such that $I = I(n)$.

The probability tree for Example 2.11 is shown in Figure 2.7. The probability distribution over the interpretations is

I	$\{inf, pn, ang\}$	$\{inf, pn\}$	$\{inf, ang\}$	$\{inf\}$
$P(I)$	0.11	0.32	0.07	0.5

There can be more than one probability tree for a program but Vennekens et al. [2009] show that all the probability trees for the program define the same probability distribution over interpretations. So we can speak of *the* probability tree for \mathcal{P} and this defines the semantics of the CP-logic program. Moreover, each program has at least one probability tree.

Vennekens et al. [2009] also show that the probability distribution defined by the LPADs semantics is the same as that defined by the CP-logic semantics. So probability trees represent an alternative definition of the DS for LPADs.

If the program contains negation, checking the truth of the body of a clause must be made with care because an atom that is currently absent from $I(n)$ may become true later. Therefore, we must make sure that for each negative literal $\sim a$ in $\text{body}(C_i)$, the positive literal a cannot be made true starting from $I(n)$.

Example 30 (CP-logic program – pneumonia [Vennekens et al., 2009]). *A patient has pneumonia. Because of pneumonia, the patient is treated. If the patient has pneumonia and is not treated, he may get fever.*

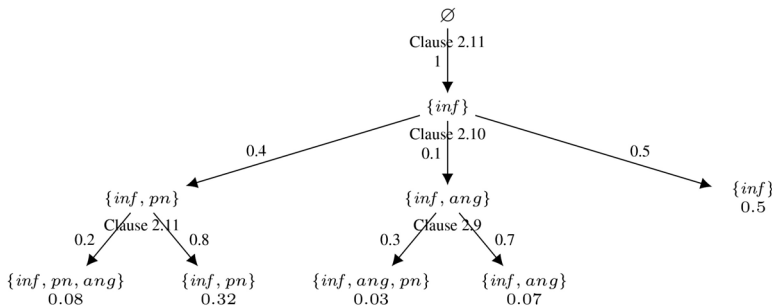


Figure 2.7 Probability tree for Example 2.11. From [Vennekens et al., 2009].

$$pneumonia. \tag{2.12}$$

$$treatment : 0.95 \leftarrow pneumonia. \tag{2.13}$$

$$fever : 0.7 \leftarrow pneumonia, \sim treatment. \tag{2.14}$$

Two probability trees for this program are shown in Figures 2.8 and 2.9. Both trees satisfy Definition 18 but define two different probability distributions. In the tree of Figure 2.8, Clause 2.14 has negative literal $\sim treatment$ in its body and is applied at a stage where $treatment$ may still become true, as happens in the level below.

In the tree of Figure 2.9, instead Clause 2.14 is applied when the only rule for $treatment$ has already fired, so in the right child of the node at the second level $treatment$ will never become true and Clause 2.14 can safely be applied.

In order to formally define this, we need the following definition that uses three-valued logic. A conjunction in three-valued logic is true or undefined if no literal in it is false.

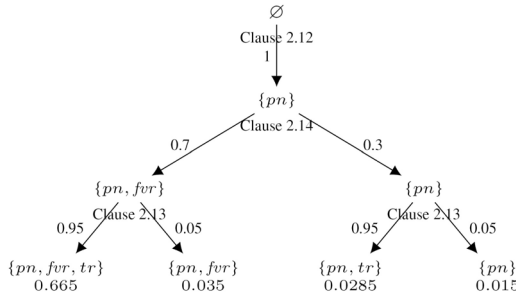


Figure 2.8 An incorrect probability tree for Example 30. From [Vennekens et al., 2009].

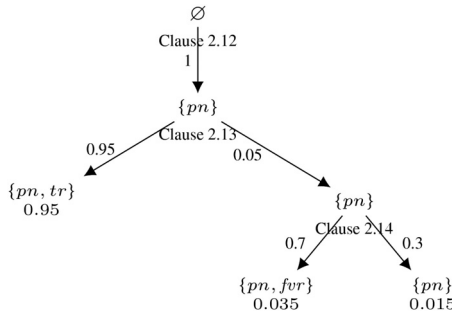


Figure 2.9 A probability tree for Example 30. From [Vennekens et al., 2009].

Definition 19 (Hypothetical derivation sequence). A hypothetical derivation sequence in a node n is a sequence $(\mathcal{I}_i)_{0 \leq i \leq n}$ of three-valued interpretations that satisfy the following properties. Initially, \mathcal{I}_0 assigns false to all atoms not in $I(n)$. For each $i > 0$, $\mathcal{I}_{i+1} = \langle I_{T,i+1}, I_{F,i+1} \rangle$ is obtained from $\mathcal{I}_i = \langle I_{T,i}, I_{F,i} \rangle$ by considering a rule R with $\text{body}(R)$ true or undefined in \mathcal{I}_i and an atom a in its head that is false in \mathcal{I} . Then $I_{T,i+1} = I_{T,i}$ and $I_{F,i+1} = I_{F,i} \setminus \{a\}$.

Every hypothetical derivation sequence reaches the same limit. For a node n in a probabilistic tree, we denote this unique limit as $\mathcal{I}(n)$. It represents the set of atoms that might still become true; in other words, all the atoms in the false part of $\mathcal{I}(n)$ will never become true and so they can be considered as false.

The definition of probability tree of a program with negation becomes the following.

Definition 20 (Probability tree – general case). A probability tree T for a program \mathcal{P} is a tree

- satisfying the conditions of Definition 18, and
- for each node n and associated clause C_i , for each negative literal $\sim a$ in $\text{body}(C_i)$, $a \in I_F$ with $\mathcal{I}(n) = \langle I_T, I_F \rangle$.

All the probability trees according for the program according to Definition 20 establish the same probability distribution over interpretations.

It can be shown that the set of false atoms of the limit of the hypothetical derivation sequence is equal to the greatest fixpoint of the operator $OpFalse_{\mathcal{I}}^P$ (see Definition 2) with $\mathcal{I} = \langle I(n), \emptyset \rangle$ and P a program that contains, for each rule

$$h_1 : \Pi_1 ; \dots ; h_m : \Pi_n \leftarrow B$$

of \mathcal{P} , the rules

$$h_1 \leftarrow B.$$

...

$$h_m \leftarrow B.$$

In other words, if $\mathcal{I}(n) = \langle I_T, I_F \rangle$ and $\text{gfp}(OpFalse_{\mathcal{I}}^P) = F$, then $I_F = F$.

In fact, for the body of a clause to be true or undefined in $\mathcal{I}_i = \langle I_{T,i}, I_{F,i} \rangle$, each positive literal a must be absent from $I_{F,i}$ and each negative literal $\sim a$ must be such that a is absent from $I_{T,i}$, which are the complementary conditions in the definition of the operator $OpFalse_{\mathcal{I}}^P(Fa)$.

On the other hand, the generation of a child n' of a node n using a rule C_i that adds an atom a to $I(n)$ can be seen as part of an application of $OpTrue_{I(n)}^P$. So there is a strong connection between CP-logic and the WFS.

In the trees of Figures 2.8 and 2.9, the child $n = \{pn\}$ of the root has $I_F = \emptyset$, so Clause 2.14 cannot be applied as $treatment \notin I_F$ and the only tree allowed by Definition 20 is that of Figure 2.9.

The semantics of CP-logic satisfies these causality principles:

- The *principle of universal causation* states that all changes to the state of the domain must be triggered by a causal law whose precondition is satisfied.
- The *principle of sufficient causation* states that if the precondition to a causal law is satisfied, then the event that it triggers must eventually happen.

and therefore the logic is particularly suitable for representing causation.

Moreover, CP-logic satisfies the *temporal precedence assumption* that states that a rule R will not fire until its precondition is in its final state. In other words, a rule fires only when the causal process that determines whether its precondition holds is fully finished. This is enforced by the treatment of negation of CP-logic.

There are CP-logic programs that do not admit any probability tree, as the following example shows.

Example 31 (Invalid CP-logic program [Vennekens et al., 2009]). *In a two-player game, white wins if black does not win and black wins if white does not win:*

$$win(white) \leftarrow \sim win(black). \quad (2.15)$$

$$win(black) \leftarrow \sim win(white). \quad (2.16)$$

At the root of the probability tree for this program, both Clauses 2.15 and 2.16 have their body true but they cannot fire as I_F for the root is \emptyset . So the root is a leaf where however two rules have their body true, thus violating the condition of Definition 18 that requires that leaves cannot be associated with rules.

This theory is problematic from a causal point of view, as it is impossible to define a process that follows the causal laws. Therefore, we want to exclude these cases and consider only *valid* CP-theories.

Definition 21 (Valid CP-theory). *A CP-theory is valid if it has at least one probability tree.*

The equivalence of the LPADs and CP-logic semantics is also carried to the general case of programs with negation: the probability tree of a valid CP-theory defines the same distribution as that defined by interpreting the program as an LPAD.

However, there are sound LPADs that are not valid CP-theories. Recall that a sound LPAD is one where each possible world has a two-valued WFM.

Example 32 (Sound LPAD – invalid CP-theory Vennekens et al. [2009]).

Consider the program

$$p : 0.5 ; q : 0.5 \leftarrow r.$$

$$r \leftarrow \sim p.$$

$$r \leftarrow \sim q.$$

Such a program has no probability tree, so it is not a valid CP-theory. Its possible worlds are

$$\{p \leftarrow r; r \leftarrow \sim p; r \leftarrow \sim q\}$$

and

$$\{q \leftarrow r; r \leftarrow \sim p; r \leftarrow \sim q\}$$

that both have total WFMs, $\{r, p\}$ and $\{r, q\}$, respectively, so the LPAD is sound.

In fact, it is difficult to imagine a causal process for this program.

Therefore, LPADs and CP-logic have some differences but these arise only in corner cases, so sometimes CP-logic and LPADs are used as a synonyms. This also shows that clauses in LPADs can be assigned in many cases a causal interpretation.

The equivalence of the semantics implies that, for a valid CP-theory, the leaves of the probability tree are associated with the WFMs of the possible world obtained by considering all the clauses used in the path from the root to the leaf with the head selected according to the choice of child. If the program is deterministic, the only leaf is associated with the total-well founded model of the program.

2.9 Semantics for Non-Sound Programs

In Section 2.2, we considered only sound programs, those for which every world has a two-valued WFM. In this way, we avoid non-monotonic aspects of the program and we deal with uncertainty only by means of probability theory.

When a program is not sound in fact, assigning a semantics to probabilistic logic programs is not obvious, as the next example shows.

Example 33 (Insomnia [Cozman and Mauá, 2017]). *Consider the program*
 $sleep \leftarrow \sim work, \sim insomnia.$
 $work \leftarrow \sim sleep.$
 $\alpha :: insomnia.$

*This program has two worlds, w_1 containing *insomnia* and w_2 not containing it. The first has the single stable model and total WFM*

$$I_1 = \{insomnia, \sim sleep, \sim work\}$$

The latter has two stable models

$$I_2 = \{insomnia, \sim sleep, work\}$$

$$I_3 = \{insomnia, sleep, \sim work\}$$

*and a WFM I_2 where *insomnia* is true and the other two atoms are undefined.*

*If we ask for the probability of *sleep*, the first world, w_1 , with probability α , surely doesn't contribute. We are not sure instead what to do with the second, as *sleep* is included in only one of the two stable models and it is undefined in the WFM.*

To handle programs like the above, Hadjichristodoulou and Warren [2012] proposed the WFS for probabilistic logic programs where a program defines a probability distribution over WFMs rather than two-valued models. This induces a probability distribution over random variables associated with atoms that are, however, three-valued instead of Boolean.

An alternative approach, the *credal semantics* [Cozman and Mauá, 2017], sees such programs as defining a set of probability measures over the interpretations. The name derives from the fact that sets of probability distributions are often called *credal sets*.

The semantics considers programs syntactically equal to ProbLog (i.e., non-probabilistic rules and probabilistic facts) and generates worlds as in ProbLog. The semantics requires that each world of the program has at least one stable models. Such programs are called *consistent*.

A program then defines a set of probability distributions over the set of all possible two-valued interpretations of the program. Each distribution P in the set is called a *probability model* and must satisfy two conditions:

1. every interpretation I for which $P(I) > 0$ must be a stable model of the world w_σ that agrees with I on the truth value of the probabilistic facts;

2. the sum of the probabilities of the stable models of w must be equal to $P(\sigma)$.

A set of distributions is obtained because we do not fix how the probability mass $P(\sigma)$ of a world w_σ is distributed over its stable models when there is more than one. We indicate with \mathbf{P} the set of probability models and call it the *credal semantics* of the program. Given a probability model, we can compute the probability of a query q as for the Distribution Semantics (DS), by summing $P(I)$ for all the interpretations I where q is true.

In this case, given a query q , we are interested in the *lower and upper probabilities* of q defined as

$$\begin{aligned}\underline{P}(q) &= \inf_{P \in \mathbf{P}} P(q) \\ \overline{P}(q) &= \sup_{P \in \mathbf{P}} P(q)\end{aligned}$$

If we are also given evidence e , Cozman and Mauá [2017] define *lower and upper conditional probabilities* as

$$\begin{aligned}\underline{P}(q|e) &= \inf_{P \in \mathbf{P}, P(e) > 0} P(q) \\ \overline{P}(q|e) &= \sup_{P \in \mathbf{P}, P(e) > 0} P(q)\end{aligned}$$

and leave them undefined when $P(e) = 0$ for all $P \in \mathbf{P}$.

Example 34 (Insomnia – continued – [Cozman and Mauá, 2017]). *Consider again the program of Example 33. A probability model that assigns the following probabilities to the models of the program*

$$\begin{aligned}P(I_1) &= \alpha \\ P(I_2) &= \gamma(1 - \alpha) \\ P(I_3) &= (1 - \gamma)(1 - \alpha)\end{aligned}$$

for $\gamma \in [0, 1]$, satisfies the two conditions of the semantics, and thus belongs to \mathbf{P} . The elements of \mathbf{P} are obtained by varying γ .

Considering the query *sleep*, we can easily see that $\underline{P}(\text{sleep} = \text{true}) = 0$ and $\overline{P}(\text{sleep} = \text{true}) = 1 - \alpha$.

With the semantics of [Hadjichristodoulou and Warren, 2012] instead, we have

$$\begin{aligned}P(I_1) &= \alpha \\ P(I_2) &= 1 - \alpha\end{aligned}$$

so

$$\begin{aligned} P(\text{sleep} = \text{true}) &= 0 \\ P(\text{sleep} = \text{false}) &= \alpha \\ P(\text{sleep} = \text{undefined}) &= 1 - \alpha. \end{aligned}$$

Example 35 (Barber paradox – [Cozman and Mauá, 2017]). *The barber paradox was introduced by Russell [1967]. If the village barber shaves all, and only, those in the village who don't shave themselves, does the barber shave himself?*

A probabilistic version of this paradox can be encoded with the program

shaves(X, Y) ← barber(X), villager(Y), ~shaves(Y, Y).

villager(a).

barber(b).

0.5 :: villager(b).

and the query shaves(b, b).

The program has two worlds, w_1 and w_2 , the first not containing the fact villager(b) and the latter containing it. The first world has a single stable model $I_1 = \{\text{villager}(a), \text{barber}(b), \text{shaves}(b, a)\}$ that is also the total WFM. In the latter world, the rule has an instance that can be simplified to $\text{shaves}(b, b) \leftarrow \sim \text{shaves}(b, b)$. Since it contains a loop through an odd number of negations, the world has no stable model and the three-valued WFM:

$$\mathcal{I}_2 = \{\text{villager}(a), \text{barber}(b), \text{shaves}(b, a), \sim \text{shaves}(a, a), \sim \text{shaves}(a, b)\}.$$

So the program is not consistent and the credal semantics is not defined for it, while the semantics of [Hadjichristodoulou and Warren, 2012] is still defined and would yield

$$\begin{aligned} P(\text{shaves}(b, b) = \text{true}) &= 0.5 \\ P(\text{shaves}(b, b) = \text{undefined}) &= 0.5 \end{aligned}$$

The WFS for probabilistic logic programs assigns a semantics to more programs. However, it introduces the truth value *undefined* that expresses uncertainty and, since probability is used as well to deal with uncertainty, some confusion may arise. For example, one may ask what is the value of $(q = \text{true} | e = \text{undefined})$. If $e = \text{undefined}$ means that we don't know anything about e , then $P(q = \text{true} | e = \text{undefined})$ should be equal to $P(q = \text{true})$ but this is not true in general. The credal semantics avoids these problems by considering only two truth values.

Cozman and Mauá [2017] show that the set \mathbf{P} is the set of all probability measures that dominate an infinitely monotone Choquet capacity.

An *infinitely monotone Choquet capacity* is a function \underline{P} from an algebra Ω on a set W to the real interval $[0, 1]$ such that

1. $\underline{P}(W) = 1 - \underline{P}(\emptyset) = 1$, and
2. for any $\omega_1, \dots, \omega_n \subseteq \Omega$,

$$\underline{P}(\cup_i \omega_i) \geq \sum_{J \subseteq \{1, \dots, n\}} (-1)^{|J|+1} \underline{P}(\cap_{j \in J} \omega_j) \quad (2.17)$$

Infinitely monotone Choquet capacity is a generalization of finitely additive probability measures: the latter are special cases of the first where Equation (2.17) holds with equality. In fact, the right member of Equation (2.17) is an application of the inclusion–exclusion principle that gives the probability of the union of non-disjoint sets. Infinitely monotone Choquet capacities also appear as belief functions of Dempster–Shafer theory [Shafer, 1976].

Given an infinitely monotone Choquet capacity \underline{P} , we can construct the set of measures $D(\underline{P})$ that dominate \underline{P} as

$$D(\underline{P}) = \{P \mid \forall \omega \in \Omega : P(\omega) \geq \underline{P}(\omega)\}$$

We say that \underline{P} *generates* the credal set $D(\underline{P})$ and we call $D(\underline{P})$ an *infinitely monotone credal set*. It is possible to show that the lower probability of $D(\underline{P})$ is exactly the generating infinitely monotone Choquet capacity: $\underline{P}(\omega) = \inf_{P \in D(\underline{P})} P(\omega)$.

Infinitely monotone credal sets are closed and convex. Convexity here means that if P_1 and P_2 are in the credal set, then $\alpha P_1 + (1 - \alpha)P_2$ is also in the credal set for $\alpha \in [0, 1]$. Given a consistent program, its credal semantics is thus a closed and convex set of probability measures.

Moreover, given a query q , we have

$$\underline{P}(q) = \sum_{w \in W, AS(w) \subseteq J_q} P(\sigma) \quad \bar{P}(q) = \sum_{w \in W, AS(w) \cap J_q \neq \emptyset} P(\sigma)$$

where J_q is the set of interpretations where q is true and $AS(w)$ is the set of stable models of world w_σ .

The lower and upper conditional probabilities of a query q are given by:

$$\underline{P}(q|e) = \frac{\underline{P}(q, e)}{\underline{P}(q, e) + \bar{P}(\neg q, e)} \quad (2.18)$$

$$\bar{P}(q|e) = \frac{\bar{P}(q, e)}{\bar{P}(q, e) + \underline{P}(\neg q, e)} \quad (2.19)$$

2.10 KBMC Probabilistic Logic Programming Languages

In this section, we present three examples of KBMC languages: Bayesian Logic Programs (BLPs), CLP(BN), and the Prolog Factor Language (PFL).

2.10.1 Bayesian Logic Programs

BLPs [Kersting and De Raedt, 2001] use logic programming to compactly encode a large BN. In BLPs, each ground atom represents a (not necessarily Boolean) random variable and the clauses define the dependencies between ground atoms. A clause of the form

$$a|a_1, \dots, a_m$$

indicates that, for each of its groundings $(a|a_1, \dots, a_m)\theta$, $a\theta$ has $a_1\theta, \dots, a_m\theta$ as parents. The domains and CPTs for the ground atom/random variables are defined in a separate portion of the model. In the case where a ground atom $a\theta$ appears in the head of more than one clause, a *combining rule* is used to obtain the overall CPT from those given by individual clauses.

For example, in the Mendelian genetics program of Example 25, the dependency that gives the value of the color gene on chromosome 1 of a plant as a function of the color genes of its mother can be expressed as

$$cg(X,1)|mother(Y,X),cg(Y,1),cg(Y,2).$$

where the domain of atoms built on predicate $cg/2$ is $\{p,w\}$ and the domain of $mother(Y,X)$ is Boolean. A suitable CPT should then be defined that assigns equal probability to the alleles of the mother to be inherited by the plant.

Various learning systems use BLPs as the representation language: RBLP [Revoredo and Zaverucha, 2002; Paes et al., 2005], PFORTE [Paes et al., 2006], and SCOOBY [Kersting and De Raedt, 2008].

2.10.2 CLP(BN)

In a CLP(BN) program [Costa et al., 2003], logical variables can be random. Their domain, parents, and CPTs are defined by the program. Probabilistic dependencies are expressed by means of constraints as in Constraint Logic Programming (CLP):

```
{ Var = Function with p(Values, Dist) }
{ Var = Function with p(Values, Dist, Parents) }
```

The first form indicates that the logical variable `Var` is random with domain `Values` and CPT `Dist` but without parents; the second form defines a random variable with parents. In both forms, `Function` is a term over logical variables that is used to parameterize the random variable: a different random variable is defined for each instantiation of the logical variables in the term. For example, the following snippet from a school domain:

```
course_difficulty(CKey, Dif) :-
  { Dif = difficulty(CKey) with p([h,m,l],
    [0.25, 0.50, 0.25]) }.
```

defines the random variable `Dif` with values `h`, `m`, and `l` representing the difficulty of the course identified by `CKey`. There is a different random variable for every instantiation of `CKey`, i.e., for each course. In a similar manner, the intelligence `Int` of a student identified by `SKey` is given by

```
student_intelligence(SKey, Int) :-
  { Int = intelligence(SKey) with p([h, m, l],
    [0.5,0.4,0.1]) }.
```

Using the above predicates, the following snippet predicts the grade received by a student when taking the exam of a course.

```
registration_grade(Key, Grade) :-
  registration(Key, CKey, SKey),
  course_difficulty(CKey, Dif),
  student_intelligence(SKey, Int),
  { Grade = grade(Key) with p(['A','B','C','D'],
    % h/h h/m h/l m/h m/m m/l l/h l/m l/l
    [0.20,0.70,0.85,0.10,0.20,0.50,0.01,0.05,0.10,
    % 'A'
    0.60,0.25,0.12,0.30,0.60,0.35,0.04,0.15,0.40,
    % 'B'
    0.15,0.04,0.02,0.40,0.15,0.12,0.50,0.60,0.40,
    % 'C'
    0.05,0.01,0.01,0.20,0.05,0.03,0.45,0.20,0.10],
    % 'D'
    [Int,Dif]) }.
```

Here `Grade` indicates a random variable parameterized by the identifier `Key` of a registration of a student to a course. The code states that there

is a different random variable `Grade` for each student's registration in a course and each such random variable has possible values `'A'`, `'B'`, `'C'` and `'D'`. The actual value of the random variable depends on the intelligence of the student and on the difficulty of the course, that are thus its parents. Together with facts for `registration/3` such as

```
registration(r0,c16,s0).  registration(r1,c10,s0).
registration(r2,c57,s0).  registration(r3,c22,s1).
.....
```

the code defines a BN with a `Grade` random variable for each registration. `CLP(BN)` is implemented as a library of YAP Prolog. The library performs query answering by constructing the sub-network that is relevant to the query and then applying a BN inference algorithm.

The unconditional probability of a random variable can be computed by simply asking a query to the YAP command line.

The answer will be a probability distribution over the values of the logical variables of the query that represent random variables, as in

```
?- registration_grade(r0,G).
    p(G=a)=0.4115,
    p(G=b)=0.356,
    p(G=c)=0.16575,
    p(G=d)=0.06675 ?
```

Conditional queries can be posed by including in the query ground atoms representing the evidence.

For example, the probability distribution of the grades of registration `r0` given that the intelligence of the student is high (`h`) is given by

```
?- registration_grade(r0,G),
    student_intelligence(s0,h).
    p(G=a)=0.6125,
    p(G=b)=0.305,
    p(G=c)=0.0625,
    p(G=d)=0.02 ?
```

In general, CLP provides a useful tool for Probabilistic Logic Programming (PLP), as is testified by the proposals `clp(pdf(Y))` [Angelopoulos, 2003, 2004] and Probabilistic Constraint Logic Programming (PCLP) [Michels et al., 2015], see Section 4.5.

2.10.3 The Prolog Factor Language

The PFL [Gomes and Costa, 2012] is an extension of Prolog for representing first-order probabilistic models.

Most graphical models such as BNs and MNs concisely represent a joint distribution by encoding it as a set of factors. The probability of a set of variables \mathbf{X} taking value \mathbf{x} can be expressed as the product of n factors as:

$$P(\mathbf{X} = \mathbf{x}) = \frac{\prod_{i=1, \dots, n} \phi_i(\mathbf{x}_i)}{Z}$$

where \mathbf{x}_i is a sub-vector of \mathbf{x} on which the i -th factor depends and Z is the normalization constant. Often, in a graphical model, the same factors appear repeatedly in the network, and thus we can parameterize these factors in order to simplify the representation.

A Parameterized Random Variables (PRVs) is a logical atom representing a set of random variables, one for each of its possible ground instantiations. We indicate PRV as X, Y, \dots and vectors of PRVs as $\mathbf{X}, \mathbf{Y}, \dots$

A *parametric factor* or *parfactor* [Kisynski and Poole, 2009b] is a triple $\langle \mathcal{C}, \mathcal{V}, F \rangle$ where \mathcal{C} is a set of inequality constraints on parameters (logical variables), \mathcal{V} is a set of PRVs and F is a factor that is a function from the Cartesian product of ranges of PRVs in \mathcal{V} to real values. A parfactor is also represented as $F(\mathcal{V})|\mathcal{C}$ or $F(\mathcal{V})$ if there are no constraints. A constrained PRV is of the form $V|\mathcal{C}$, where $V = p(X_1, \dots, X_n)$ is a non-ground atom and \mathcal{C} is a set of constraints on logical variables $\mathbf{X} = \{X_1, \dots, X_n\}$. Each constrained PRV represents the set of random variables $\{P(\mathbf{x})|\mathbf{x} \in \mathcal{C}\}$, where \mathbf{x} is the tuple of constants (x_1, \dots, x_n) . Given a (constrained) PRV V , we use $RV(V)$ to denote the set of random variables it represents. Each ground atom is associated with one random variable, which can take any value in $range(V)$.

The PFL extends Prolog to support probabilistic reasoning with parametric factors. A PFL factor is a parfactor of the form

$$Type \mathbf{F} ; \phi ; \mathcal{C},$$

where *Type* refers to the type of the network over which the parfactor is defined (*bayes* for directed networks or *markov* for undirected ones); \mathbf{F} is a sequence of Prolog goals each defining a PRV under the constraints in \mathcal{C} (the arguments of the factor). If \mathbf{L} is the set of all logical variables in \mathbf{F} , then \mathcal{C} is a list of Prolog goals that impose bindings on \mathbf{L} (the successful substitutions for

the goals in \mathcal{C} are the valid values for the variables in \mathbf{L}). ϕ is the table defining the factor in the form of a list of real values. By default, all random variables are Boolean but a different domain may be defined. Each parfactor represents the set of its groundings. To ground a parfactor, all variables of \mathbf{L} are replaced with the values permitted by constraints in \mathcal{C} . The set of ground factors defines a factorization of the joint probability distribution over all random variables.

Example 36 (PFL program). *The following PFL program is inspired by the workshop attributes problem of [Milch et al., 2008]. It models the organization of a workshop where a number of people have been invited. `series` indicates whether the workshop is successful enough to start a series of related meetings while `attends(P)` indicates whether person P attends the workshop.*

This problem can be modeled by a PFL program such as

```
bayes series, attends(P); [0.51, 0.49, 0.49, 0.51];
    [person(P)].
bayes attends(P), at(P,A); [0.7, 0.3, 0.3, 0.7];
    [person(P), attribute(A)].
```

A workshop becomes a series because people attend. People attend the workshop depending on the workshop's attributes such as location, date, fame of the organizers, etc. The probabilistic atom `at(P,A)` represents whether person P attends because of attribute A .

The first PFL factor has the random variables `series` and `attends(P)` as arguments (both Boolean), `[0.51, 0.49, 0.49, 0.51]` as table and the list `[person(P)]` as constraint.

Since KBMC languages are defined on the basis of a translation to graphical models, translations can be built between PLP languages under the DS and KBMC languages. The first have the advantage that they have a semantics that can be understood in logical terms, without necessarily referring to an underlying graphical model.

2.11 Other Semantics for Probabilistic Logic Programming

Here we briefly discuss a few examples of PLP frameworks that don't follow the distribution semantics. Our goal in this section is simply to give the flavor of other possible approaches; a complete account of such frameworks is beyond the scope of this book.

2.11.1 Stochastic Logic Programs

Stochastic Logic Programs (SLPs) [Muggleton et al., 1996; Cussens, 2001] are logic programs with parameterized clauses which define a distribution over refutations of goals. The distribution provides, by marginalization, a distribution over variable bindings for the query. SLPs are a generalization of stochastic grammars and hidden Markov models.

An SLP S is a definite logic program where some of the clauses are of the form $p : C$ where $p \in \mathbb{R}, p \geq 0$, and C is a definite clause. Let $n(S)$ be the definite logic program obtained by removing the probability labels. A *pure* SLP is an SLP where all clauses have probability labels. A *normalized* SLP is one where probability labels for clauses whose heads share the same predicate symbol sum to one.

In pure SLPs, each SLD derivation for a query q is assigned a real label by multiplying the labels of each individual derivation step. The label of a derivation step where the selected atom unifies with the head of clause $p_i : C_i$ is p_i . The probability of a successful derivation from q is the label of the derivation divided by the sum of the labels of all the successful derivations. This forms a distribution over successful derivations from q .

The probability of an instantiation $q\theta$ is the sum of the probabilities of the successful derivations that produce $q\theta$. It can be shown that the probabilities of all the atoms for a predicate q that succeed in $n(S)$ sum to one, i.e., S defines a probability distribution over the success set of q in $n(S)$.

In impure SLPs, the unparameterized clauses are seen as non-probabilistic domain knowledge acting as constraints. Derivations are identified with the set of the parameterized clauses they use. In this way, derivations that differ only on the unparameterized clauses form an equivalence class.

In practice, SLPs define probability distributions over the children of nodes of the SLD tree for a query: a derivation step $u \rightarrow v$ that connects node u with child node v is assigned a probability $P(v|u)$. This induces a probability distributions over paths from the root to the leaves of the SLD tree and in turn over answers for the query.

Given their similarity with stochastic grammars and hidden Markov models, SLPs are particularly suitable for representing these kinds of models. They differ from the DS because they define a probability distribution over instantiations of the query, while the DS usually defines a distribution over the truth values of ground atoms.

Example 37 (Probabilistic context-free grammar – SLP). *Consider the probabilistic context free grammar:*

$$0.2 : S \rightarrow aS$$

$$0.2 : S \rightarrow bS$$

$$0.3 : S \rightarrow a$$

$$0.3 : S \rightarrow b$$

The SLP

$$0.2 : s([a|R]) \leftarrow s(R).$$

$$0.2 : s([b|R]) \leftarrow s(R).$$

$$0.3 : s([a]).$$

$$0.3 : s([b]).$$

defines a distribution over the values of S in $s(S)$ that is the same as the one defined by the probabilistic context-free grammar above. For example, $P(s([a, b])) = 0.2 \cdot 0.3 = 0.6$ according to the program and $P(ab) = 0.2 \cdot 0.3 = 0.6$ according to the grammar.

Various approaches have been proposed for learning SLPs. Muggleton [2000a,b] proposed to use an Inductive Logic Programming (ILP) system, Progol [Muggleton, 1995], for learning the structure of the programs, and a second phase where the parameters are tuned using a generalization of relative frequency.

Parameters are also learned by means of optimization in failure-adjusted maximization [Cussens, 2001; Angelopoulos, 2016] and by solving algebraic equations [Muggleton, 2003].

2.11.2 ProPPR

ProPPR [Wang et al., 2015] is an extension of SLPs that that is related to Personalized PageRank (PPR) [Page et al., 1999].

ProPPR extends SLPs in two ways. The first is the method for computing the labels of the derivation steps. A derivation step $u \rightarrow v$ is not simply assigned the parameter associated with the clause used in the step. Instead, the label of the derivation step, $P(v|u)$ is computed using a log-linear model $P(v|u) \propto \exp(\mathbf{w} \cdot \phi_{u \rightarrow v})$ where \mathbf{w} is a vector of real-valued weights and $\phi_{u \rightarrow v}$ is a 0/1 vector of “features” that depend on the clause being used. The features are user defined and the association between clauses and features is indicated using annotations.

Example 38 (ProPPR program). *The ProPPR program [Wang et al., 2015]*

$$\text{about}(X, Z) \leftarrow \text{handLabeled}(X, Z). \quad \#base$$

$$\text{about}(X, Z) \leftarrow \text{sim}(X, Y), \text{about}(Y, Z). \quad \#prop$$

$sim(X, Y) \leftarrow link(X, Y).$	$\#sim, link$
$sim(X, Y) \leftarrow hasWord(X, W), hasWord(Y, W),$	
$linkedBy(X, Y, W).$	$\#sim, word$
$linkedBy(X, Y, W).$	$\#by(W)$

can be used to compute the topic of web pages on the basis of possible hand labeling or similarity with other web pages. Similarity is defined as well in a probabilistic way depending on the links and words between the two pages.

Clauses are annotated with a list of atoms (indicated after the # symbol) that may contain variables from the head of clauses. In the example, the third clause is annotated with the list of atoms $sim, link$ while the last clause is annotated by the atom $by(W)$. Each grounding of each atom in the list stands for a different feature, so for example $sim, link$, and $by(sprinter)$ stand for three different features. The vector $\phi_{u \rightarrow v}$ is obtained by assigning value 1 to the features associated with the atoms in the annotation of the clause used for the derivation step $u \rightarrow v$ and value 0 otherwise. If the atoms contain variables, these are shared with the head of the clause and are grounded with the values of the clause instantiation used in $u \rightarrow v$.

So a ProPPR program is defined by an annotated program plus values for the weights w . This annotation approach considerably increases the flexibility of SLP labels: ProPPR annotations can be shared across clauses and can yield labels that depend on the particular clause grounding that is used by the derivation step. An SLP is a ProPPR program where each clause has a different annotation consisting of an atom without arguments.

The second way in which ProPPR extend SLPs consists in the addition of edges to the SLD tree: an edge is added (a) from every solution leaf to itself; and (b) from every node to the start node.

The procedure for assigning probabilities to queries of SLP can then be applied to the resulting graph. The self-loop links heuristically upweight solution nodes and the restart links make SLP's graph traversal a PPR procedure [Page et al., 1999]: a PageRank can be associated with each node, representing the probability that a random walker starting from the root arrives in that node.

The restart links favor the results of short proofs: if the restart probability is α for every node u , then the probability of reaching any node at depth d is bounded by $(1 - \alpha)^d$.

Parameter learning for ProPPR is performed in [Wang et al., 2015] by stochastic gradient descent.

2.12 Other Semantics for Probabilistic Logics

In this section, we discuss semantics for probabilistic logic languages that are not based on logic programming.

2.12.1 Nilsson's Probabilistic Logic

Nilsson's probabilistic logic [Nilsson, 1986] takes an approach for combining logic and probability that is different from the DS: while the first considers sets of distributions, the latter computes a single distribution over possible worlds. In Nilsson's logic, a *probabilistic interpretation* Pr defines a probability distribution over the set of interpretations $Int2$. The *probability of a logical formula* F according to Pr , denoted $Pr(F)$, is the sum of all $Pr(I)$ such that $I \in Int2$ and $I \models F$. A *probabilistic knowledge base* \mathcal{K} is a set of probabilistic formulas of the form $F \geq p$. A probabilistic interpretation Pr *satisfies* $F \geq p$ iff $Pr(F) \geq p$. Pr *satisfies* \mathcal{K} , or Pr is a *model* of \mathcal{K} , iff Pr satisfies all $F \geq p \in \mathcal{K}$. $Pr(F) \geq p$ is a *tight logical consequence* of \mathcal{K} iff p is the infimum of $Pr(F)$ in the set of all models Pr of \mathcal{K} . Computing tight logical consequences from probabilistic knowledge bases can be done by solving a linear optimization problem.

With Nilsson's logic, the consequences that can be obtained from logical formulas differ from those of the DS. Consider a ProbLog program (see Section 2.1) composed of the facts $0.4 :: c(a)$ and $0.5 :: c(b)$, and a probabilistic knowledge base composed of $c(a) \geq 0.4$ and $c(b) \geq 0.5$. For the DS, $P(c(a) \vee c(b)) = 0.7$, while with Nilsson's logic, the lowest p such that $Pr(c(a) \vee c(b)) \geq p$ holds is 0.5. This difference is due to the fact that, while Nilsson's logic makes no assumption about the independence of the statements, in the DS, the probabilistic axioms are considered as independent. While independencies can be encoded in Nilsson's logic by carefully choosing the values of the parameters, reading off the independencies from the theories becomes more difficult.

However, the assumption of independence of probabilistic axioms does not restrict expressiveness as shown in Section 2.6.

2.12.2 Markov Logic Networks

A Markov Logic Network (MLN) is a first-order logical theory in which each sentence is associated with a real-valued weight. An MLN is a template for generating MNs. Given sets of constants defining the domains of the logical variables, an MLN defines an MN that has a Boolean node for each ground

atom and edges connecting the atoms appearing together in a grounding of a formula. MLNs follow the KBMC approach for defining a probabilistic model [Wellman et al., 1992; Bacchus, 1993]. The probability distribution encoded by an Markov Logic Network (MLN) is

$$P(\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_{f_i \in M} w_i n_i(\mathbf{x})\right)$$

where \mathbf{x} is a joint assignment of truth value to all atoms in the Herbrand base (finite because of no function symbols), M is the MLN, f_i is the i -th formula in M , w_i is its weight, $n_i(\mathbf{x})$ is the number of groundings of formula f_i that are satisfied in \mathbf{x} , and Z is a normalization constant.

Example 39 (Markov Logic Network). *The following MLN encodes a theory on the intelligence of friends and on the marks people get:*

```

1.5 Intelligent(x) => GoodMarks(x)
1.1 Friends(x,y) => (Intelligent(x) <=>
                      Intelligent(y))
    
```

The first formula gives a positive weight to the fact that if someone is intelligent, then he gets good marks in the exams he takes. The second formula gives a positive weight to the fact that friends have similar intelligence: in particular, the formula states that if x and y are friends, then x is intelligent if and only if y is intelligent, so they are either both intelligent or both not intelligent.

If the domain contains two individuals, Anna and Bob, indicated with A and B , we get the ground MN of Figure 2.10.

2.12.2.1 Encoding Markov Logic Networks with Probabilistic Logic Programming

It is possible to encode MNs and MLNs with LPADs. The encoding is based on the BN that is equivalent to the MN as discussed in Section 1.6: an MN

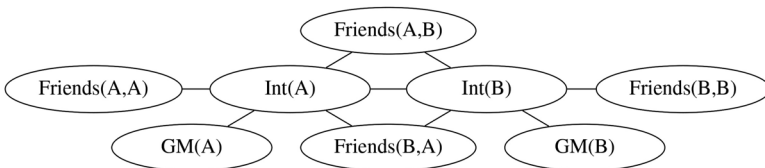


Figure 2.10 Ground Markov network for the MLN of Example 39.

factor can be represented with an extra node in the equivalent BN that is always observed. In order to model MLN formulas with LPADs, we can add an extra atom $clause_i(\mathbf{X})$ for each formula $F_i = w_i C_i$ where w_i is the weight associated with C_i and \mathbf{X} is the vector of variables appearing in C_i . Then, when we ask for the probability of query q given evidence e , we have to ask for the probability of q given $e \wedge ce$, where ce is the conjunction of the groundings of $clause_i(\mathbf{X})$ for all values of i .

Clause C_i must be transformed into a Disjunctive Normal Form (DNF) formula $C_{i1} \vee \dots \vee C_{im_i}$, where the disjuncts are mutually exclusive and the LPAD should contain the clauses

$$clause_i(\mathbf{X}) : e^\alpha / (1 + e^\alpha) \leftarrow C_{ij}$$

for all j in $1, \dots, n_i$, where $1 + e^\alpha \geq \max_{\mathbf{x}_i} \phi(\mathbf{x}_i) = \max\{1, e^\alpha\}$. Similarly, $\neg C_i$ must be transformed into a DNF $D_{i1} \vee \dots \vee D_{im_i}$ and the LPAD should contain the clauses

$$clause_i(\mathbf{X}) : 1 / (1 + e^\alpha) \leftarrow D_{il}$$

for all l in $1, \dots, m_i$.

Moreover, for each predicate p/n , we should add the clause

$$p(\mathbf{X}) : 0.5.$$

to the program, assigning *a priori* uniform probability to every ground atom.

Alternatively, if α is negative, e^α will be smaller than 1 and $\max_{\mathbf{x}_i} \phi(\mathbf{x}_i) = 1$. So we can use the two probability values e^α and 1 with the clauses

$$clause_i(\mathbf{X}) : e^\alpha \leftarrow C_{ij}.$$

$$clause_i(\mathbf{X}) \leftarrow D_{il}.$$

This solution has the advantage that some clauses are non-probabilistic, reducing the number of random variables. If α is positive in the formula αC , we can consider the equivalent formula $-\alpha \neg C$.

The transformation above is illustrated by the following example. Given the MLN

```
1.5 Intelligent(x) => GoodMarks(x)
1.1 Friends(x,y) => (Intelligent(x) <=> Intelligent(y))
```

the first formula is translated to the clauses:

```
clause1(X):0.8175 :- \+intelligent(X).
clause1(X):0.1824 :- intelligent(X),
                    \+good_marks(X).
clause1(X):0.8175 :- intelligent(X),good_marks(X).
```

where $0.8175 = e^{1.5}/(1 + e^{-1.5})$ and $0.1824 = 1/(1 + e^{-1.5})$.

The second formula is translated to the clauses

```
clause2(X,Y):0.7502 :- \+friends(X,Y).
clause2(X,Y):0.7502 :- friends(X,Y),
                    intelligent(X),
                    intelligent(Y).
clause2(X,Y):0.7502 :- friends(X,Y),
                    \+intelligent(X),
                    \+intelligent(Y).
clause2(X,Y):0.2497 :- friends(X,Y),
                    intelligent(X),
                    \+intelligent(Y).
clause2(X,Y):0.2497 :- friends(X,Y),
                    \+intelligent(X),
                    intelligent(Y).
```

where $0.7502 = e^{1.1}/(1 + e^{1.1})$ and $0.2497 = 1/(1 + e^{1.1})$.

A priori we have a uniform distribution over student intelligence, good marks, and friendship:

```
intelligent(_):0.5.
good_marks(_):0.5.
friends(_,_):0.5.
```

and there are two students:

```
student(anna).
student(bob).
```

We have evidence that Anna is friend with Bob and Bob is intelligent. The evidence must also include the truth of all groundings of the *clause_i* predicates:

```
evidence_mln :- clause1(anna),clause1(bob),
                clause2(anna,anna),clause2(anna,bob),
                clause2(bob,anna),clause2(bob,bob).
ev_intelligent_bob_friends_anna_bob :-
    intelligent(bob),friends(anna,bob),
    evidence_mln.
```

The probability that Anna gets good marks given the evidence is thus

$$P(\text{good_marks}(\text{anna}) \mid \text{ev_intelligent_bob_friends_anna_bob})$$

while the prior probability of Anna getting good marks is given by

$$P(\text{good_marks}(\text{anna})).$$

The probability resulting from the first query is higher ($P = 0.733$) than the second query ($P = 0.607$), since it is conditioned to the evidence that Bob is intelligent and Anna is his friend.

In the alternative transformation, the first MLN formula is translated to:

```
clause1(X) :- \+intelligent(X).
clause1(X):0.2231 :- intelligent(X), \+good_marks(X).
clause1(X) :- intelligent(X), good_marks(X).
```

where $0.2231 = e^{-1.5}$.

MLN formulas can also be added to a regular probabilistic logic program. In this case, their effect is equivalent to a soft form of evidence, where certain worlds are weighted more than others. This is the same as soft evidence in Figaro [Pfeffer, 2016]. MLN hard constraints, i.e., formulas with an infinite weight, can instead be used to rule out completely certain worlds, those violating the constraint. For example, given hard constraint C equivalent to the disjunction $C_{i1} \vee \dots \vee C_{in_i}$, the LPAD should contain the clauses

$$\text{clause}_i(\mathbf{X}) \leftarrow C_{ij}$$

for all j , and the evidence should contain $\text{clause}_i(\mathbf{x})$ for all groundings \mathbf{x} of \mathbf{X} . In this way, the worlds that violate C are ruled out.

2.12.3 Annotated Probabilistic Logic Programs

In Annotated Probabilistic Logic Programming (APLP) [Ng and Subrahmanian, 1992], program atoms are annotated with intervals that can be interpreted probabilistically. An example rule in this approach is:

$$a : [0.75, 0.85] \leftarrow b : [1, 1], c : [0.5, 0.75]$$

that states that the probability of a is between 0.75 and 0.85 if b is certainly true and the probability of c is between 0.5 and 0.75. The probability interval of a conjunction or disjunction of atoms is defined using a *combinator* to

construct the tightest bounds for the formula. For instance, if d is annotated with $[l_d, h_d]$ and e with $[l_e, h_e]$, the probability of $e \wedge d$ is annotated with

$$[\max(0, l_d + l_e - 1), \min(h_d, h_e)].$$

Using these combinators, an inference operator and fixpoint semantics is defined for positive Datalog programs. A model theory is obtained for such programs by considering the annotations as constraints on acceptable probabilistic worlds: an APLP thus describes a family of probabilistic worlds.

APLPs have the advantage that deduction is of low complexity, as the logic is truth-functional, i.e., the probability of a query can be computed directly using combinators. The corresponding disadvantages are that APLPs may be inconsistent if they are not carefully written, and that the use of the above combinators may quickly lead to assigning overly slack probability intervals to certain atoms. These aspects are partially addressed by hybrid APLPs Dekhtyar and Subrahmanian [2000], which allow different flavors of combinators based on, e.g., independence or mutual exclusivity of given atoms.

3

Semantics with Function Symbols

When a program contains variables, function symbols, and at least one constant, its grounding is infinite. In this case, the number of atomic choices in a selection that defines a world is countably infinite and there is an uncountably infinite number of worlds. The probability of each individual world is given by an infinite product. We recall the following result from [Knopp, 1951, page 218].

Lemma 1 (Infinite Product). *If $p_i \in [0, b]$ for all $i = 1, 2, \dots$ with $b \in [0, 1]$, then the infinite product $\prod_{i=1}^{\infty} p_i$ converges to 0.*

Each factor in the infinite product giving the probability of a world is bounded away from one, i.e., it belongs to $[0, b]$ for $b \in [0, 1)$. To see this, it is enough to pick b as the maximum of all the probabilistic parameters that appear in the program. This is possible if the program does not have flexible probabilities or probabilities that depend on values computed during program execution.

So if the program does not contain flexible probabilities, the probability of each individual world is zero and the semantics of Section 2.2 is not well-defined [Riguzzi, 2016].

Example 40 (Program with infinite set of worlds). *Consider the ProbLog program*

$$\begin{aligned} p(0) &\leftarrow u(0). \\ p(s(X)) &\leftarrow p(X), u(X). \\ t &\leftarrow \sim s. \\ s &\leftarrow r, q. \\ q &\leftarrow u(X). \\ F_1 &= a :: u(X). \\ F_2 &= b :: r. \end{aligned}$$

The set of worlds is infinite and uncountable. In fact, each selection can be represented as a countable sequence of atomic choices of which the first

involves fact f_2 , the second $f_1/\{X/0\}$, the third $f_1/\{X/s(0)\}$, and so on. The set of selections can be shown uncountable by Cantor's diagonal argument. Suppose the set of selections is countable. Then the selections could be listed in order, suppose from top to bottom. Suppose the atomic choices of each selection are listed from left to right. We can pick a composite choice that differs from the first selection in the first atomic choice (if (f_2, \emptyset, k) is the first atomic choice of the first selection, pick $(f_2, \emptyset, 1 - k)$), from the second selection in the second atomic choice (similar to the case of the first atomic choice), and so on. In this way, we have obtained a selection that is not present in the list because it differs from each selection in the list for at least an atomic choice. So it is not possible to list the selections in order against the hypothesis.

Example 41 (Game of dice). Consider the game of dice proposed in [Vennekens et al., 2004]: the player repeatedly throws a six-sided die. When the outcome is six, the game stops. A ProbLog version of this game where the die has three sides is:

$$F_1 = 1/3 :: \text{one}(X).$$

$$F_2 = 1/2 :: \text{two}(X).$$

$$\text{on}(0, 1) \leftarrow \text{one}(0).$$

$$\text{on}(0, 2) \leftarrow \sim \text{one}(0), \text{two}(0).$$

$$\text{on}(0, 3) \leftarrow \sim \text{one}(0), \sim \text{two}(0).$$

$$\text{on}(s(X), 1) \leftarrow \text{on}(X, -), \sim \text{on}(X, 3), \text{one}(s(X)).$$

$$\text{on}(s(X), 2) \leftarrow \text{on}(X, -), \sim \text{on}(X, 3), \sim \text{one}(s(X)), \text{two}(s(X)).$$

$$\text{on}(s(X), 3) \leftarrow \text{on}(X, -), \sim \text{on}(X, 3), \sim \text{one}(s(X)), \sim \text{two}(s(X)).$$

If we add the clauses

$$\text{at_least_once_1} \leftarrow \text{on}(-, 1).$$

$$\text{never_1} \leftarrow \sim \text{at_least_once_1}.$$

we can ask for the probability that at least once the die landed on face 1 and that the die never landed on face 1. As in Example 40, this program has an infinite and uncountable set of worlds.

3.1 The Distribution Semantics for Programs with Function Symbols

We now present the definition of the DS for ProbLog programs with function symbols following [Poole, 1997]. The semantics for a probabilistic logic program \mathcal{P} with function symbols of [Poole, 1997] is given by defining a finitely additive probability measure μ over an algebra $\Omega_{\mathcal{P}}$ on the set of worlds $W_{\mathcal{P}}$.

We first need some definitions. The *set of worlds* ω_κ κ compatible with a composite choice is $\omega_\kappa = \{w_\sigma \in W_{\mathcal{P}} \mid \kappa \subseteq \sigma\}$. Thus, a composite choice identifies a set of worlds. For programs without function symbols, $P(\kappa) = \sum_{w \in \omega_\kappa} P(w)$, where

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

For program with function symbols $\sum_{w \in \omega_\kappa} P(w)$ may not be defined as ω_κ may uncountable and $P(w) = 0$. However, $P(\kappa)$ is still well defined. Let us call it μ so $\mu(\kappa) = P(\kappa)$.

Given a *set of composite choices* K , the *set of worlds* ω_K compatible with K is $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$. Two composite choices κ_1 and κ_2 are *incompatible* if their union is not consistent. A set K of composite choices is *pairwise incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2$ implies that κ_1 and κ_2 are incompatible.

Regardless of whether a probabilistic logic program has a finite number of worlds or not, obtaining pairwise incompatible sets of composite choices is an important problem. This is because for program without function symbols, the *probability of a pairwise incompatible set K of composite choices* can be defined as $P(K) = \sum_{\kappa \in K} P(\kappa)$ which is easily computed. For programs with function symbols, $P(K)$ is still well defined provided that K , is countable. Let us call it μ so $\mu(K) = P(K)$. Two sets K_1 and K_2 of composite choices are *equivalent* if they correspond to the same set of worlds: $\omega_{K_1} = \omega_{K_2}$.

One way to assign probabilities to a set K of composite choices is to construct an equivalent set that is pairwise incompatible; such a set can be constructed through the technique of *splitting*. More specifically, if $f\theta$ is an instantiated fact and κ is a composite choice that does not contain an atomic choice (f, θ, k) for any k , the *split* of κ on $f\theta$ is the set of composite choices $S_{\kappa, f\theta} = \{\kappa \cup \{(f, \theta, 0)\}, \kappa \cup \{(f, \theta, 1)\}\}$. It is easy to see that κ and $S_{\kappa, f\theta}$ identify the same set of possible worlds, i.e., that $\omega_\kappa = \omega_{S_{\kappa, f\theta}}$, and that $S_{\kappa, f\theta}$ is pairwise incompatible. The technique of splitting composite choices on formulas is used for the following result [Poole, 2000].

Theorem 4 (Existence of a pairwise incompatible set of composite choices [Poole, 2000]). *Given a finite set K of composite choices, there exists a finite set K' of pairwise incompatible composite choices such that K and K' are equivalent.*

Proof. Given a finite set of composite choices K , there are two possibilities to form a new set K' of composite choices so that K and K' are equivalent:

1. **Removing dominated elements:** if $\kappa_1, \kappa_2 \in K$ and $\kappa_1 \subset \kappa_2$, let $K' = K \setminus \{\kappa_2\}$.
2. **Splitting elements:** if $\kappa_1, \kappa_2 \in K$ are compatible (and neither is a superset of the other), there is a $(f, \theta, k) \in \kappa_1 \setminus \kappa_2$. We replace κ_2 by the split of κ_2 on $f\theta$. Let $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2, f\theta}$.

In both cases, $\omega_K = \omega_{K'}$. If we repeat this two operations until neither is applicable, we obtain a splitting algorithm (see Algorithm 1) that terminates because K is a finite set of composite choices. The resulting set K' is pairwise incompatible and is equivalent to the original set. \square

Algorithm 1 Function SPLIT: Splitting Algorithm.

```

1: function SPLIT( $K$ )
2:   loop
3:     if  $\exists \kappa_1, \kappa_2 \in K$  such that  $\kappa_1 \subset \kappa_2$  then
4:        $K \leftarrow K \setminus \{\kappa_2\}$ 
5:     else
6:       if  $\exists \kappa_1, \kappa_2 \in K$  compatible then
7:         choose  $(f, \theta, k) \in \kappa_1 \setminus \kappa_2$ 
8:          $K \leftarrow K \setminus \{\kappa_2\} \cup S_{\kappa_2, F\theta}$ 
9:       else
10:        return  $K$ 
11:      end if
12:    end if
13:  end loop
14: end function

```

Theorem 5 (Equivalence of the probability of two equivalent pairwise incompatible finite set of finite composite choices [Poole, 1993a]). *If K_1 and K_2 are both pairwise incompatible finite sets of finite composite choices such that they are equivalent, then $P(K_1) = P(K_2)$.*

Proof. Consider the set D of all instantiated facts $f\theta$ that appear in an atomic choice in either K_1 or K_2 . This set is finite. Each composite choice in K_1 and K_2 has atomic choices for a subset of D . For both K_1 and K_2 , we repeatedly replace each composite choice κ of K_1 and K_2 with its split $S_{\kappa, f_i\theta_j}$ on an $f_i\theta_j$ from D that does not appear in κ . This procedure does not change the total probability as the probabilities of $(f_i, \theta_j, 0)$ and $(f_i, \theta_j, 1)$ sum to 1.

At the end of this procedure, the two sets of composite choices will be identical. In fact, any difference can be extended into a possible world belonging to ω_{K_1} but not to ω_{K_2} or vice versa. \square

For a ProbLog program \mathcal{P} , we can thus define a unique finitely additive probability measure $\mu_{\mathcal{P}}^F : \Omega_{\mathcal{P}} \rightarrow [0, 1]$ where $\Omega_{\mathcal{P}}$ is defined as the set of sets of worlds identified by finite sets of finite composite choices: $\Omega_{\mathcal{P}} = \{\omega_K \mid K \text{ is a finite set of finite composite choices}\}$.

Theorem 6 (Algebra of a program). *$\Omega_{\mathcal{P}}$ is an algebra over $W_{\mathcal{P}}$.*

Proof. We need to prove that $\Omega_{\mathcal{P}}$ respects the three conditions of Definition 7. $W_{\mathcal{P}} = \omega_K$ with $K = \{\emptyset\}$. The complement ω_K^c of ω_K where K is a finite set of finite composite choice is $\omega_{\overline{K}}$ where \overline{K} is a finite set of finite composite choices. In fact, \overline{K} can be obtained with the function $\text{DUALS}(K)$ of [Poole, 2000] shown in Algorithm 2 for the case of ProbLog. Such a function performs Reiter's hitting set algorithm over K , generating an element κ of \overline{K} by picking an atomic choice (f, θ, k) from each element of K and inserting in κ the atomic choice $(f, \theta, 1-k)$. After this process is performed in all possible ways, inconsistent sets of atom choices are removed obtaining \overline{K} . Since the possible choices of atomic choices are finite, so is \overline{K} . Finally, closure under finite union holds since the union of ω_{K_1} with ω_{K_2} is equal to $\omega_{K_1 \cup K_2}$ for the definition of ω_K . \square

Algorithm 2 Function DUALS: Duals computation.

```

1: function DUALS( $K$ )
2:   suppose  $K = \{\kappa_1, \dots, \kappa_n\}$ 
3:    $D_0 \leftarrow \{\emptyset\}$ 
4:   for  $i \leftarrow 1 \rightarrow n$  do
5:      $D_i \leftarrow \{d \cup \{(f, \theta, 1-k)\} \mid d \in D_{i-1}, (f, \theta, k) \in \kappa_i\}$ 
6:     remove inconsistent elements from  $D_i$ 
7:     remove any  $\kappa$  from  $D_i$  if  $\exists \kappa' \in D_i$  such that  $\kappa' \subset \kappa$ 
8:   end for
9:   return  $D_n$ 
10: end function
    
```

The corresponding measure $\mu_{\mathcal{P}}^F$ is defined by $\mu_{\mathcal{P}}^F(\omega_K) = \mu(K')$ where K' is a pairwise incompatible set of composite choices equivalent to K .

Theorem 7 (Finitely additive probability space of a program). *The triple $\langle W_{\mathcal{P}}, \Omega_{\mathcal{P}}, \mu_{\mathcal{P}}^F \rangle$ with*

$$\mu_{\mathcal{P}}^F(\omega_K) = \mu(K')$$

where K' is a pairwise incompatible set of composite choices equivalent to K , is a finitely additive probability space according to Definition 11.

Proof. $\mu_{\mathcal{P}}^F(\omega_{\{\emptyset\}})$ is equal to 1. Moreover, $\mu_{\mathcal{P}}^F(\omega_K) \geq 0$ for all K and if $\omega_{K_1} \cap \omega_{K_2} = \emptyset$ and $K'_1 (K'_2)$ is pairwise incompatible and equivalent to $K_1 (K_2)$, then $K'_1 \cup K'_2$ is pairwise incompatible and

$$\begin{aligned} \mu_{\mathcal{P}}^F(\omega_{K_1 \cup K_2}) &= \sum_{\kappa \in K'_1 \cup K'_2} P(\kappa) = \sum_{\kappa_1 \in K'_1} P(\kappa_1) + \sum_{\kappa_2 \in K'_2} P(\kappa_2) = \\ &\mu_{\mathcal{P}}^F(\omega_{K_1}) + \mu_{\mathcal{P}}^F(\omega_{K_2}). \end{aligned}$$

□

Given a query q , a composite choice κ is an *explanation* for q if $\forall w \in \omega_{\kappa} : w \models q$. A set K of composite choices is *covering* with respect to q if every world in which q is true belongs to ω_K .

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

$$Q(w) = \begin{cases} 1 & \text{if } w \models q \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

If q has a finite set K of finite explanations such that K is covering then $Q^{-1}(\{1\}) = \{w | w \in W_{\mathcal{P}} \wedge w \models q\} = \omega_K \in \Omega_{\mathcal{P}}$ so Q is measurable. Therefore, Q is a random variable whose distribution is defined by $P(Q = 1)$ ($P(Q = 0)$ is given by $1 - P(Q = 1)$). We indicate $P(Q = 1)$ with $P(q)$ and we say that $P(q)$ is *finitely well-defined* for the distribution semantics. A program \mathcal{P} is *finitely well-defined* if the probability of all ground atoms in the grounding of \mathcal{P} is finitely well-defined.

Example 42 (Covering set of explanations for Example 40). *Consider the program of Example 40. The set $K = \{\kappa\}$ with*

$$\kappa = \{(f_1, \{X/0\}, 1), (f_1, \{X/s(0)\}, 1)\}$$

is a pairwise incompatible finite set of finite explanations that are covering for the query $p(s(0))$. Then $P(p(s(0)))$ is finitely well-defined and $P(p(s(0))) = P(\kappa) = a^2$.

Example 43 (Covering set of explanations for Example 41). *Now consider Example 41. The set $K = \{\kappa_1, \kappa_2\}$ with*

$$\begin{aligned} \kappa_1 &= \{(f_1, \{X/0\}, 1), (f_1, \{X/s(0)\}, 1)\} \\ \kappa_2 &= \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), (f_1, \{X/s(0)\}, 1)\} \end{aligned}$$

is a pairwise incompatible finite set of finite explanations that are covering for the query $\text{on}(s(0), 1)$. Then $P(\text{on}(s(0), 1))$ is finitely well-and

$$P(\text{on}(s(0), 1)) = P(K) = 1/3 \cdot 1/3 + 2/3 \cdot 1/2 \cdot 1/3 = 2/9.$$

3.2 Infinite Covering Set of Explanations

In this section, we go beyond [Poole, 1997] and we remove the requirement of the finiteness of the covering set of explanations and of each explanation for a query q [Riguzzi, 2016].

Example 44 (Pairwise incompatible covering set of explanations for Example 40). *In Example 40, the query s has the pairwise incompatible covering set of explanations*

$$K^s = \{\kappa_0^s, \kappa_1^s, \dots\}$$

with

$$\kappa_i^s = \{(f_2, \emptyset, 1), (f_1, \{X/0\}, 0), \dots, \\ (f_1, \{X/s^{i-1}(0)\}, 0), (f_1, \{X/s^i(0)\}, 1)\}$$

where $s^i(0)$ is the term where the functor s is applied i times to 0. So K^s is countable and infinite. A pairwise incompatible covering set of explanations for t is

$$K^t = \{\{(f_2, \emptyset, 0)\}, \kappa^t\}$$

where κ^t is the infinite composite choice

$$\kappa^t = \{(f_2, \emptyset, 1), (f_1, \{X/0\}, 0), (f_1, \{X/s(0)\}, 0), \dots\}$$

Example 45 (Pairwise incompatible covering set of explanations for Example 41). *In Example 41, the query `at_least_once_1` has the pairwise incompatible covering set of explanations*

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

with

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

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

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

with

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

For a probabilistic logic program \mathcal{P} , we can define the probability measure $\mu_{\mathcal{P}} : \Omega_{\mathcal{P}} \rightarrow [0, 1]$ where $\Omega_{\mathcal{P}}$ is defined as the set of sets of worlds identified by countable sets of countable composite choices: $\Omega_{\mathcal{P}} = \{\omega_K \mid K \text{ is a countable set of countable composite choices}\}$.

Before showing that $\Omega_{\mathcal{P}}$ is a σ -algebra, we need some definitions and results regarding sequences of sets. For any sequence of sets $\{A_n \mid n \geq 1\}$, define [Chow and Teicher, 2012, page 2]

$$\begin{aligned} \underline{\lim}_{n \rightarrow \infty} A_n &= \bigcup_{n=1}^{\infty} \bigcap_{k=n}^{\infty} A_k \\ \overline{\lim}_{n \rightarrow \infty} A_n &= \bigcap_{n=1}^{\infty} \bigcup_{k=n}^{\infty} A_k \end{aligned}$$

Note that [Chow and Teicher, 2012, page 2]

$$\begin{aligned} \overline{\lim}_{n \rightarrow \infty} A_n &= \{a \mid a \in A_n \text{ i.o.}\} \\ \underline{\lim}_{n \rightarrow \infty} A_n &= \{a \mid a \in A_n \text{ for all but a finite number of indices } n\} \end{aligned}$$

where i.o. denotes infinitely often. The two definitions differ because an element a of $\overline{\lim}_{n \rightarrow \infty} A_n$ may be absent from A_n for an infinite number of indices n , provided that there is a disjoint, infinite set of indices n for which $a \in A_n$. For each $a \in \underline{\lim}_{n \rightarrow \infty} A_n$, instead, there is a $m \geq 1$ such that $\forall n \geq m, a \in A_n$.

Then $\underline{\lim}_{n \rightarrow \infty} A_n \subseteq \overline{\lim}_{n \rightarrow \infty} A_n$. If $\underline{\lim}_{n \rightarrow \infty} A_n = \overline{\lim}_{n \rightarrow \infty} A_n = A$, then A is called the *limit of the sequence* and we write $A = \lim_{n \rightarrow \infty} A_n$.

A sequence $\{A_n | n \geq 1\}$ is *increasing* if $A_{n-1} \subseteq A_n$ for all $n = 2, 3, \dots$. If a sequence $\{A_n | n \geq 1\}$ is increasing, the limit $\lim_{n \rightarrow \infty} A_n$ exists and is equal to $\bigcup_{n=1}^{\infty} A_n$ [Chow and Teicher, 2012, page 3].

Lemma 2 (σ -algebra of a Program). $\Omega_{\mathcal{P}}$ is a σ -algebra over $W_{\mathcal{P}}$.

Proof. $W_{\mathcal{P}} \in \Omega_{\mathcal{P}}$ is true as in the algebra case. To see that the complement ω_K^c of ω_K is in $\Omega_{\mathcal{P}}$, let us prove that the dual \overline{K} of K is a countable set of countable composite choices and then that $\omega_K^c = \omega_{\overline{K}}$. Let us consider first the case where K is finite, i.e., let K be $K_n = \{\kappa_1, \dots, \kappa_n\}$. We will prove the thesis by induction. In the base case, if $K_1 = \{\kappa_1\}$, then we can obtain \overline{K}_1 by picking each atomic choice (f, θ, k) of κ_1 and inserting in \overline{K}_1 the composite choice $\{(f, \theta, 1 - k)\}$. As there is a finite or countable number of atomic choices in κ_1 , \overline{K}_1 is a finite or countable set of composite choices each with one atomic choice.

In the inductive case, assume that $K_{n-1} = \{\kappa_1, \dots, \kappa_{n-1}\}$ and that \overline{K}_{n-1} is a finite or countable set of composite choices. Let $K_n = K_{n-1} \cup \{\kappa_n\}$ and $\overline{K}_{n-1} = \{\kappa'_1, \kappa'_2, \dots\}$. We can obtain \overline{K}_n by picking each κ'_i and each atomic choice (f, θ, k) of κ_n . If $(f, \theta, k) \in \kappa'_i$, we discard κ'_i , else if $(f, \theta, k') \in \kappa'_i$ with $k' \neq k$, we insert κ'_i in \overline{K}_n . Otherwise, we generate the composite choice κ''_i where $\kappa''_i = \kappa'_i \cup \{(f, \theta, 1 - k)\}$ and insert it in \overline{K}_n . Doing this for all atomic choices (f, θ, k) , in κ_n generates a finite set of finite composite choices if κ_n is finite and a countable number of finite composite choices if κ_n is countable. Doing this for all κ'_i , we obtain that \overline{K}_n is a countable union of countable sets which is a countable set [Cohn, 2003, page 3]. $\omega_K^c = \omega_{\overline{K}}$ because all composite choices of \overline{K} are incompatible with each world of ω_K , as they are incompatible with each composite choice of K . So $\omega_K^c \in \Omega_{\mathcal{P}}$.

If K is not finite, then let $K = \{\kappa_1, \kappa_2, \dots\}$. Consider the subsets K_n of the form $K_n = \{\kappa_1, \dots, \kappa_n\}$. Using the construction above build \overline{K}_n for all n and consider the set $\underline{\lim}_{n \rightarrow \infty} \overline{K}_n$ and $\overline{\lim}_{n \rightarrow \infty} \overline{K}_n$. Consider a κ' that belongs to $\overline{\lim}_{n \rightarrow \infty} \overline{K}_n$. Suppose $\kappa' \in \overline{K}_j$ and $\kappa' \notin \overline{K}_{j+1}$. This means that κ' was removed because $\kappa_{j+1} \subseteq \kappa'$ or because it was replaced by an extension of it. Then κ' will never be re-added to a \overline{K}_n with $n > j+1$ because otherwise ω_{K_n} and $\omega_{\overline{K}_n}$ would have a non-empty intersection. So for a composite choice κ' to appear infinitely often, there must exist an integer $m \geq 1$ such that $\kappa' \in \overline{K}_n$ for all $n \geq m$. In other words, κ' belongs to $\underline{\lim}_{n \rightarrow \infty} \overline{K}_n$. Therefore,

$\underline{\lim}_{n \rightarrow \infty} \overline{K}_n = \overline{\lim}_{n \rightarrow \infty} \overline{K}_n = \lim_{n \rightarrow \infty} \overline{K}_n$. Let us call \overline{K} this limit. \overline{K} can thus be expressed as $\bigcup_{n=1}^{\infty} \bigcap_{k=n}^{\infty} \overline{K}_k$.

$\bigcap_{k=n}^{\infty} \overline{K}_k$ is countable as it is a countable intersection of countable sets. So \overline{K} is countable as it is a countable union of countable sets. Moreover, each composite choice of \overline{K} is incompatible with each composite choice of K . In fact, let κ' be an element of \overline{K} and let $m \geq 1$ be the smallest integer such that $\kappa' \in \overline{K}_n$ for all $n \geq m$. Then κ' is incompatible with all composite choices of K_n for $n \geq m$ by construction. Moreover, it was obtained by extending a composite choice κ'' from \overline{K}_{m-1} that was incompatible with all composite choices from K_{m-1} . As κ' is an extension of κ'' , it is also incompatible with all elements of K_{m-1} . So $\omega_K^c = \omega_{\overline{K}}$ and $\omega_K^c \in \Omega_{\mathcal{P}}$.

Closure under countable union is true as in the algebra case. \square

Given $K = \{\kappa_1, \kappa_2, \dots\}$ where the κ_i s may be infinite, consider the sequence $\{K_n | n \geq 1\}$ where $K_n = \{\kappa_1, \dots, \kappa_n\}$. Since K_n is an increasing sequence, the limit $\lim_{n \rightarrow \infty} K_n$ exists and is equal to K . Let us build a sequence $\{K'_n | n \geq 1\}$ as follows: $K'_1 = \{\kappa_1\}$ and K'_n is obtained by the union of K'_{n-1} with the splitting of each element of K'_{n-1} with κ_n . By induction, it is possible to prove that K'_n is pairwise incompatible and equivalent to K_n .

For each K'_n , we can compute $\mu(K'_n)$, noting that $\mu(\kappa) = 0$ for infinite composite choices. Let us consider the limit $\lim_{n \rightarrow \infty} \mu(K'_n)$.

Lemma 3 (Existence of the limit of the measure of countable union of countable composite choices). $\lim_{n \rightarrow \infty} \mu(K'_n)$ exists.

Proof. We can see $\mu(K'_n)$ for $n = 1, \dots$ as the partial sums of a series. A non-decreasing series converges if the partial sums are bounded from above [Brannan, 2006, page 92], so, if we prove that $\mu(K'_n) \geq \mu(K'_{n-1})$ and that $\mu(K'_n)$ is bounded by 1, the lemma is proved. Remove from K'_n the infinite composite choices, as they have measure 0. Let \mathcal{D}_n be a ProbLog program containing a fact $\Pi_i :: f_i\theta$ for each instantiated facts $f_i\theta$ that appears in an atomic choice of K'_n . Then $\mathcal{D}_{n-1} \subseteq \mathcal{D}_n$. The triple $(W_{\mathcal{D}_n}, \Omega_{\mathcal{D}_n}, \mu)$ is a finitely additive probability space (see Section 2.2), so $\mu(K'_n) \leq 1$. Moreover, since $\omega_{K'_{n-1}} \subseteq \omega_{K'_n}$, then $\mu(K'_n) \geq \mu(K'_{n-1})$. \square

We can now define the probability space of a program.

Theorem 8 (Probability space of a program). *The triple $\langle W_{\mathcal{P}}, \Omega_{\mathcal{P}}, \mu_{\mathcal{P}} \rangle$ with*

$$\mu_{\mathcal{P}}(\omega_K) = \lim_{n \rightarrow \infty} \mu(K'_n)$$

where $K = \{\kappa_1, \kappa_2, \dots\}$ and K'_n is a pairwise incompatible set of composite choices equivalent to $\{\kappa_1, \dots, \kappa_n\}$, is a probability space according to Definition 10.

Proof. (μ -1) and (μ -2) hold as for the finite case. For (μ -3), let

$$O = \{\omega_{L_1}, \omega_{L_2}, \dots\}$$

be a countable set of subsets of $\Omega_{\mathcal{P}}$ such that the ω_{L_i} s are the set of worlds compatible with countable sets of countable composite choices L_i s and are pairwise disjoint. Let L'_i be the pairwise incompatible set equivalent to L_i and let \mathcal{L} be $\bigcup_{i=1}^{\infty} L'_i$. Since the ω_{L_i} s are pairwise disjoint, then \mathcal{L} is pairwise incompatible. \mathcal{L} is countable as it is a countable union of countable sets. Let \mathcal{L} be $\{\kappa_1, \kappa_2, \dots\}$ and let K'_n be $\{\kappa_1, \dots, \kappa_n\}$. Then

$$\mu_{\mathcal{P}}(O) = \lim_{n \rightarrow \infty} \mu(K'_n) = \lim_{n \rightarrow \infty} \sum_{\kappa \in K'_n} \mu(\kappa) = \sum_{i=1}^{\infty} \mu(\kappa) = \sum_{\kappa \in \mathcal{L}} \mu(\kappa).$$

Since $\sum_{i=1}^{\infty} \mu(\kappa)$ is convergent and a sum of non-negative terms, it is also absolutely convergent and its terms can be rearranged [Knopp, 1951, Theorem 4, page 142]. We thus get

$$\mu_{\mathcal{P}}(O) = \sum_{\kappa \in \mathcal{L}} \mu(\kappa) = \sum_{n=1}^{\infty} \mu(L'_n) = \sum_{n=1}^{\infty} \mu_{\mathcal{P}}(\omega_{L_n}).$$

□

For a probabilistic logic program \mathcal{P} and a ground atom q with a countable set K of explanations such that K is covering for q , then $\{w | w \in W_{\mathcal{P}} \wedge w \models q\} = \omega_K \in \Omega_{\mathcal{P}}$. So function Q of Equation (3.1) is a random variable.

Again we indicate $P(Q = 1)$ with $P(q)$ and we say that $P(q)$ is *well-defined* for the distribution semantics. A program \mathcal{P} is *well-defined* if the probability of all ground atoms in the grounding of \mathcal{P} is well-defined.

Example 46 (Probability of the query for Example 40). *Consider Example 44. The explanations in K^s are pairwise incompatible, so the probability of s can be computed as*

$$P(s) = ba + ba(1 - a) + ba(1 - a)^2 + \dots = \frac{ba}{1 - (1 - a)} = b.$$

since the sum is a geometric series. K^t is also pairwise incompatible, and $P(\kappa^t) = 0$ so $P(t) = 1 - b + 0 = 1 - b$ which is what we intuitively expect.

Example 47 (Probability of the query for Example 41). *In Example 45, the explanations in K^+ are pairwise incompatible, so the probability of the query `at_least_once_1` is given by*

$$\begin{aligned} P(\text{at_least_once_1}) &= \frac{1}{3} + \frac{2}{3} \cdot \frac{1}{2} \cdot \frac{1}{3} + \left(\frac{2}{3} \cdot \frac{1}{2}\right)^2 \cdot \frac{1}{3} + \dots \\ &= \frac{1}{3} + \frac{1}{9} + \frac{1}{27} \dots \\ &= \frac{\frac{1}{3}}{1 - \frac{1}{3}} = \frac{\frac{1}{3}}{\frac{2}{3}} = \frac{1}{2} \end{aligned}$$

since the sum is a geometric series.

For the query `never_1`, the explanations in K^- are pairwise incompatible, so the probability of `never_1` can be computed as

$$\begin{aligned} P(\text{never_1}) &= \frac{2}{3} \cdot \frac{1}{2} + \frac{2}{3} \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{1}{2} + \\ &\quad \left(\frac{2}{3} \cdot \frac{1}{2}\right)^2 \cdot \frac{2}{3} \cdot \frac{1}{2} + \dots = \\ &\quad \frac{1}{3} + \frac{1}{9} + \frac{1}{27} \dots = \frac{1}{2}. \end{aligned}$$

This is expected as `never_1` = \sim `at_least_once_1`.

We now want to show that every program is well-defined, i.e., it has a countable set of countable explanations that is covering for each query. In the following, we consider only ground programs that, however, may be countably infinite, and thus they can be the result of grounding a program with function symbols.

Given two sets of composite choices K_1 and K_2 , define the conjunction $K_1 \otimes K_2$ of K_1 and K_2 as $K_1 \otimes K_2 = \{\kappa_1 \cup \kappa_2 \mid \kappa_1 \in K_1, \kappa_2 \in K_2, \text{consistent}(\kappa_1 \cup \kappa_2)\}$. It is easy to see that $\omega_{K_1 \otimes K_2} = \omega_{K_1} \cap \omega_{K_2}$.

Similarly to [Vlasselaer et al., 2015, 2016], we define parameterized interpretations and an $IFPP^P$ operator that are generalizations of interpretations and the IFP^P operator for normal programs. Differently from [Vlasselaer et al., 2015, 2016], here parameterized interpretations associate each atom with a set of composite choices rather than with a Boolean formula.

Definition 22 (Parameterized two-valued interpretations). *A parameterized positive two-valued interpretation Tr for a ground probabilistic logic program \mathcal{P} with Herbrand base $\mathcal{B}_{\mathcal{P}}$ is a set of pairs (a, K_a) with $a \in \text{atoms}$*

and K_a a set of composite choices. A parameterized negative two-valued interpretation Fa for a ground probabilistic logic program P with Herbrand base \mathcal{B}_P is a set of pairs $(a, K_{\sim a})$ with $a \in \mathcal{B}_P$ and $K_{\sim a}$ a set of composite choices.

Parameterized two-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall (a, K_a) \in I, (a, L_a) \in J : \omega_{K_a} \subseteq \omega_{L_a}$. The least upper bound and greatest lower bound always exist and are

$$\text{lub}(X) = \{(a, \bigcup_{I \in X, (a, K_a) \in I} K_a) | a \in \mathcal{B}_P\}$$

and

$$\text{glb}(X) = \{(a, \bigotimes_{I \in X, (a, K_a) \in I} K_a) | a \in \mathcal{B}_P\}.$$

The top element \top is

$$\{(a, \{\emptyset\}) | a \in \mathcal{B}_P\}$$

and the bottom element \perp is

$$\{(a, \emptyset) | a \in \mathcal{B}_P\}.$$

Definition 23 (Parameterized three-valued interpretations). A parameterized three-valued interpretation \mathcal{I} for a ground probabilistic logic program \mathcal{P} with Herbrand base \mathcal{B}_P is a set of triples $(a, K_a, K_{\sim a})$ with $a \in \mathcal{B}_P$ and K_a and $K_{\sim a}$ sets of composite choices. A consistent parameterized three-valued interpretation \mathcal{I} is such that $\forall (a, K_a, K_{\sim a}) \in \mathcal{I} : \omega_{K_a} \cap \omega_{K_{\sim a}} = \emptyset$.

Parameterized three-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall (a, K_a, K_{\sim a}) \in I, (a, L_a, L_{\sim a}) \in J : \omega_{K_a} \subseteq \omega_{L_a}$ and $\omega_{K_{\sim a}} \subseteq \omega_{L_{\sim a}}$. The least upper bound and greatest lower bound always exist and are

$$\text{lub}(X) = \{(a, \bigcup_{I \in X, (a, K_a, K_{\sim a}) \in I} K_a, \bigcup_{I \in X, (a, K_a, K_{\sim a}) \in I} K_{\sim a}) | a \in \mathcal{B}_P\}$$

and

$$\text{glb}(X) = \{(a, \bigotimes_{I \in X, (a, K_a, K_{\sim a}) \in I} K_a, \bigotimes_{I \in X, (a, K_a, K_{\sim a}) \in I} K_{\sim a}) | a \in \mathcal{B}_P\}.$$

The top element \top is

$$\{(a, \{\emptyset\}, \{\emptyset\}) | a \in \mathcal{B}_P\}$$

and the bottom element \perp is

$$\{(a, \emptyset, \emptyset) \mid a \in \mathcal{B}_{\mathcal{P}}\}.$$

Definition 24 ($OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr)$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}(Fa)$). *For a ground program \mathcal{P} with rules \mathcal{R} and facts \mathcal{F} , a two-valued parameterized positive interpretation Tr with pairs (a, L_a) , a two-valued parameterized negative interpretation Fa with pairs $(a, M_{\sim a})$, and a three-valued parameterized interpretation \mathcal{I} with triples $(a, K_a, K_{\sim a})$, we define $OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr) = \{(a, L'_a) \mid a \in \mathcal{B}_{\mathcal{P}}\}$ where*

$$L'_a = \begin{cases} \{(a, \emptyset, 1)\} & \text{if } a \in \mathcal{F} \\ \bigcup_{a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, c_m \in \mathcal{R}} ((L_{b_1} \cup K_{b_1}) \otimes \dots \otimes (L_{b_n} \cup K_{b_n}) \otimes K_{\sim c_1} \otimes \dots \otimes K_{\sim c_m}) & \text{if } a \in \mathcal{B}_{\mathcal{P}} \setminus \mathcal{F} \end{cases}$$

and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}(Fa) = \{(a, M'_a) \mid a \in \mathcal{B}_{\mathcal{P}}\}$ where

$$M'_{\sim a} = \begin{cases} \{(a, \emptyset, 0)\} & \text{if } a \in \mathcal{F} \\ \bigotimes_{a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, c_m \in \mathcal{R}} ((M_{\sim b_1} \otimes K_{\sim b_1}) \cup \dots \cup (M_{\sim b_n} \otimes K_{\sim b_n}) \cup K_{c_1} \cup \dots \cup K_{c_m}) & \text{if } a \in \mathcal{B}_{\mathcal{P}} \setminus \mathcal{F} \end{cases}$$

Proposition 2 (Monotonicity of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$). *$OpTrueP_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$ are monotonic.*

Proof. Let us consider $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$. We have to prove that if $Tr_1 \leq Tr_2$, then $OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr_1) \leq OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr_2)$. $Tr_1 \leq Tr_2$ means that

$$\forall (a, L_a) \in Tr_1, (a, M_a) \in Tr_2 : \omega_{L_a} \subseteq \omega_{M_a}.$$

Let (a, L'_a) be the elements of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr_1)$ and (a, M'_a) the elements of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}(Tr_2)$. We have to prove that $\omega_{L'_a} \subseteq \omega_{M'_a}$.

If $a \in \mathcal{F}$, then $L'_a = M'_a = \{(a, \emptyset, 1)\}$. If $a \in \mathcal{B}_{\mathcal{P}} \setminus \mathcal{F}$, then L'_a and M'_a have the same structure. Since $\forall b \in \mathcal{B}_{\mathcal{P}} : \omega_{L_b} \subseteq \omega_{M_b}$, then $\omega_{L'_a} \subseteq \omega_{M'_a}$.

We can prove similarly that $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$ is monotonic. \square

Since $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$ are monotonic, they have a least fixpoint and a greatest fixpoint.

Definition 25 (Iterated fixed point for probabilistic programs). *For a ground program \mathcal{P} , let $IFPP^{\mathcal{P}}$ be defined as*

$$IFPP^{\mathcal{P}}(\mathcal{I}) = \{(a, K_a, K_{\sim a}) \mid (a, K_a) \in \text{lfp}(OpTrueP_{\mathcal{I}}^{\mathcal{P}}), \\ (a, K_{\sim a}) \in \text{gfp}(OpFalseP_{\mathcal{I}}^{\mathcal{P}})\}.$$

Proposition 3 (Monotonicity of $IFPP^{\mathcal{P}}$). *$IFPP^{\mathcal{P}}$ is monotonic.*

Proof. We have to prove that, if $\mathcal{I}_1 \leq \mathcal{I}_2$, then $IFPP^{\mathcal{P}}(\mathcal{I}_1) \leq IFPP^{\mathcal{P}}(\mathcal{I}_2)$. $\mathcal{I}_1 \leq \mathcal{I}_2$ means that

$$\forall (a, L_a, L_{\sim a}) \in \mathcal{I}_1, (a, M_a, M_{\sim a}) \in \mathcal{I}_2 : \omega_{L_a} \subseteq \omega_{M_a}, \omega_{L_{\sim a}} \subseteq \omega_{M_{\sim a}}.$$

Let $(a, L'_a, L'_{\sim a})$ be the elements of $IFPP^{\mathcal{P}}(\mathcal{I}_1)$ and $(a, M'_a, M'_{\sim a})$ the elements of $IFPP^{\mathcal{P}}(\mathcal{I}_2)$. We have to prove that $\omega_{L'_a} \subseteq \omega_{M'_a}$ and $\omega_{L'_{\sim a}} \subseteq \omega_{M'_{\sim a}}$. This follows from the monotonicity of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$ in \mathcal{I} , which can be proved as in Proposition 2. \square

So $IFPP^{\mathcal{P}}$ has a least fixpoint. Let $WFMP(\mathcal{P})$ denote $\text{lfp}(IFPP^{\mathcal{P}})$, and let δ the smallest ordinal such that $IFPP^{\mathcal{P}} \uparrow \delta = WFMP(\mathcal{P})$. We refer to δ as the *depth* of \mathcal{P} .

Let us now prove that $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$ and $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$ are sound.

Lemma 4 (Soundness of $OpTrueP_{\mathcal{I}}^{\mathcal{P}}$). *For a ground probabilistic logic program \mathcal{P} with probabilistic facts \mathcal{F} and rules \mathcal{R} , and a parameterized three-valued interpretation \mathcal{I} , let L_a^α be the formula associated with atom a in $OpTrueP_{\mathcal{I}}^{\mathcal{P}} \uparrow \alpha$. For every atom a , total choice σ , and iteration α , we have:*

$$w_\sigma \in \omega_{L_a^\alpha} \rightarrow WFM(w_\sigma | \mathcal{I}) \models a$$

where $w_\sigma | \mathcal{I}$ is obtained by adding the atoms a for which $(a, K_a, K_{\sim a}) \in \mathcal{I}$ and $w_\sigma \in \omega_{K_a}$ to w_σ , and by removing all the rules with a in the head for which $(a, K_a, K_{\sim a}) \in \mathcal{I}$ and $w_\sigma \in \omega_{K_{\sim a}}$.

Proof. Let us prove the lemma by transfinite induction: let us assume the thesis for all $\beta < \alpha$ and let us prove it for α . If α is a successor ordinal, then it is easily verified for $a \in \mathcal{F}$. Otherwise assume $w_\sigma \in \omega_{L_a^\alpha}$ where

$$L_a^\alpha = \bigcup_{a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, c_m \in \mathcal{R}} ((L_{b_1}^{\alpha-1} \cup K_{b_1}) \otimes \dots \otimes (L_{b_n}^{\alpha-1} \cup K_{b_n}) \otimes K_{\sim c_1} \otimes \dots \otimes K_{\sim c_m})$$

This means that there is rule $a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, c_m \in \mathcal{R}$ such that $w_\sigma \in \omega_{L_{b_i}^{\alpha-1} \cup K_{b_i}}$ for $i = 1, \dots, n$ and $w_\sigma \in \omega_{K_{\sim c_j}}$ for $j = 1, \dots, m$. By the inductive assumption and because of how $w_\sigma | \mathcal{I}$ is built, then $WFM(w_\sigma | \mathcal{I}) \models b_i$ and $WFM(w_\sigma | \mathcal{I}) \models \sim c_j$ so $WFM(w_\sigma | \mathcal{I}) \models a$.

If α is a limit ordinal, then

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

If $w_\sigma \in \omega_{L_a^\alpha}$, then there must exist a $\beta < \alpha$ such that $w_\sigma \in \omega_{L_a^\beta}$. By the inductive assumption, the hypothesis holds. \square

Lemma 5 (Soundness of $OpFalseP_{\mathcal{I}}^{\mathcal{P}}$). *For a ground probabilistic logic program \mathcal{P} with probabilistic facts \mathcal{F} and rules \mathcal{R} , and a parameterized three-valued interpretation \mathcal{I} , let $M_{\sim a}^\alpha$ be the set of composite choices associated with atom a in $OpFalseP_{\mathcal{I}}^{\mathcal{P}} \downarrow \alpha$. For every atom a , total choice σ , and iteration α , we have:*

$$w_\sigma \in \omega_{M_{\sim a}^\alpha} \rightarrow WFM(w_\sigma | \mathcal{I}) \models \sim a$$

where $w_\sigma | \mathcal{I}$ is built as in Lemma 4.

Proof. Similar to the proof of Lemma 4. \square

To prove the soundness of $IFPP^{\mathcal{P}}$, we first need two lemmas, the first regarding the model of a program obtained by a partial evaluation of the semantics and the second about the equivalence of models of instances and partial programs.

Lemma 6 (Partial evaluation). *For a ground normal logic program P and a three-valued interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ such that $\mathcal{I} \leq WFM(P)$, let $P || \mathcal{I}$ be defined as the program obtained from P by adding all atoms $a \in I_T$ and by removing all rules for atoms $a \in I_F$. Then $WFM(P) = WFM(P || \mathcal{I})$.*

Proof. We first prove that $WFM(P)$ is a fixpoint of $IFP^{P || \mathcal{I}}$. Pick an atom $a \in OpTrue_{WFM(P)}^P(I_T)$. If $a \in I_T$, then a is a fact in $P || \mathcal{I}$, so it is in $OpTrue_{WFM(P)}^{P || \mathcal{I}}(I_T)$. Otherwise, there exists a rule $a \leftarrow b_1, \dots, b_n$ in P such that b_i is true in $WFM(P)$ or $b_i \in I_T$ for $i = 1, \dots, n$. Such a rule is also in $P || \mathcal{I}$ so $a \in OpTrue_{WFM(P)}^{P || \mathcal{I}}(I_T)$.

Now pick an atom $a \in OpFalse_{WFM(P)}^P(I_F)$. If $a \in I_F$, then there are no rules for a in $P || \mathcal{I}$, so it is in $OpFalse_{WFM(P)}^{P || \mathcal{I}}(I_F)$. Otherwise, for all rules $a \leftarrow b_1, \dots, b_n$ in P , there exists a b_i such that it is false in $WFM(P)$ or $b_i \in I_F$. The set of rules for a in $P || \mathcal{I}$ is the same, so $a \in OpFalse_{WFM(P)}^{P || \mathcal{I}}(I_F)$.

We can prove similarly that $WFM(P||\mathcal{I})$ is a fixpoint of $IFPP^P$.

Since $WFM(P)$ is a fixpoint of $IFPP^{P||\mathcal{I}}$, then $WFM(P||\mathcal{I}) \leq WFM(P)$ because $WFM(P||\mathcal{I})$ is the least fixpoint of $IFPP^{P||\mathcal{I}}$. Since $WFM(P||\mathcal{I})$ is a fixpoint of $IFPP^P$, then $WFM(P) \leq WFM(P||\mathcal{I})$. So $WFM(P) = WFM(P||\mathcal{I})$. \square

Lemma 7 (Model equivalence). *Given a ground probabilistic logic program \mathcal{P} , for every total choice σ , and iteration α , we have:*

$$WFM(w_\sigma) = WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha).$$

Proof. Let $\mathcal{I}_\alpha = \langle I_T, I_F \rangle$ be a three-valued interpretation defined as $I_T = \{a|w_\sigma \in K_a^\alpha\}$ and $I_F = \{a|w_\sigma \in K_{\sim a}^\alpha\}$. Then $\forall a \in I_T : WFM(w_\sigma) \models a$ and $\forall a \in I_F : WFM(w_\sigma) \models \sim a$. So $\mathcal{I}_\alpha \leq WFM(w_\sigma)$.

Since $w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha = w_\sigma||\mathcal{I}_\alpha$, by Lemma 6

$$WFM(w_\sigma) = WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha) = WFM(w_\sigma||\mathcal{I}_\alpha). \quad \square$$

The following lemma shows that $IFPP^{\mathcal{P}}$ is sound.

Lemma 8 (Soundness of $IFPP^{\mathcal{P}}$). *For a ground probabilistic logic program \mathcal{P} with probabilistic facts \mathcal{F} and rules \mathcal{R} , let K_a^α and $K_{\sim a}^\alpha$ be the formulas associated with atom a in $IFPP^{\mathcal{P}} \uparrow \alpha$. For every atom a , total choice σ , and iteration α , we have:*

$$w_\sigma \in \omega_{K_a^\alpha} \rightarrow WFM(w_\sigma) \models a \quad (3.2)$$

$$w_\sigma \in \omega_{K_{\sim a}^\alpha} \rightarrow WFM(w_\sigma) \models \sim a \quad (3.3)$$

Proof. This is a simple consequence of Lemma 7: $w_\sigma \in \omega_{K_a^\alpha}$ means that a is a fact in $WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha)$, so $WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha) \models a$ and $WFM(w_\sigma) \models a$.

On the other hand, $w_\sigma \in \omega_{K_{\sim a}^\alpha}$ means that there are not rules for a in $WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha)$, and therefore $WFM(w_\sigma|IFPP^{\mathcal{P}} \uparrow \alpha) \models \sim a$ and $WFM(w_\sigma) \models \sim a$. \square

The following lemma shows that $IFPP^{\mathcal{P}}$ is complete.

Lemma 9 (Completeness of $IFPP^{\mathcal{P}}$). *For a ground probabilistic logic program \mathcal{P} with probabilistic facts \mathcal{F} and rules \mathcal{R} , let K_a^α and $K_{\sim a}^\alpha$ be the*

formulas associated with atom a in $IFPP^{\mathcal{P}} \uparrow \alpha$. For every atom a , total choice σ , and iteration α , we have:

$$\begin{aligned} a \in IFP^{w_\sigma} \uparrow \alpha &\rightarrow w_\sigma \in \omega_{K_a^\alpha} \\ \sim a \in IFP^{w_\sigma} \uparrow \alpha &\rightarrow w_\sigma \in \omega_{K_{\sim a}^\alpha} \end{aligned}$$

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

$$\begin{aligned} a \in IFP^{w_\sigma} \uparrow (\alpha - 1) &\rightarrow w_\sigma \in \omega_{K_a^{\alpha-1}} \\ \sim a \in IFP^{w_\sigma} \uparrow (\alpha - 1) &\rightarrow w_\sigma \in \omega_{K_{\sim a}^{\alpha-1}} \end{aligned}$$

Let us perform transfinite induction on the iterations of $OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma}$ and $OpFalse_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma}$. Let us consider a successor ordinal δ : assume that

$$\begin{aligned} a \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow (\delta - 1) &\rightarrow w_\sigma \in \omega_{L_a^{\delta-1}} \\ \sim a \in OpFalse_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \downarrow (\delta - 1) &\rightarrow w_\sigma \in \omega_{M_{\sim a}^{\delta-1}} \end{aligned}$$

where $(a, K_a^{\delta-1})$ are the elements of $OpTrue_{IFPP^{\mathcal{P}} \uparrow \alpha-1}^{\mathcal{P}} \uparrow (\delta - 1)$ and $(a, M_{\sim a}^{\delta-1})$ are the elements of $OpFalse_{IFPP^{\mathcal{P}} \uparrow \alpha-1}^{\mathcal{P}} \downarrow (\delta - 1)$. We prove that

$$\begin{aligned} a \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow \delta &\rightarrow w_\sigma \in \omega_{L_a^\delta} \\ \sim a \in OpFalse_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \downarrow \delta &\rightarrow w_\sigma \in \omega_{M_{\sim a}^\delta} \end{aligned}$$

Consider a . If $a \in \mathcal{F}$, it is easily proved.

For the other atoms, $a \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow \delta$ means that there is a rule $a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, c_m$ such that for all $i = 1, \dots, n$

$$b_i \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow (\delta - 1) \vee b_i \in IFP^{w_\sigma} \uparrow (\alpha - 1)$$

and for all $j = 1, \dots, m$, $\sim c_j \in IFP^{w_\sigma} \uparrow (\alpha - 1)$. For the inductive hypothesis, $\forall i : w_\sigma \in \omega_{L_{b_i}^{\delta-1}} \vee w_\sigma \in \omega_{K_{b_i}^{\alpha-1}}$ and $\forall j : w_\sigma \in \omega_{K_{\sim c_j}^{\alpha-1}}$ so $w_\sigma \in L_a^\delta$. Analogously for $\sim a$.

If δ is a limit ordinal, then $L_a^\delta = \bigcup_{\mu < \delta} L_a^\mu$ and $M_{\sim a}^\delta = \bigotimes_{\mu < \delta} M_{\sim a}^\mu$. If $a \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow \delta$, then there exists a $\mu < \delta$ such that

$$a \in OpTrue_{IFP^{w_\sigma} \uparrow (\alpha-1)}^{w_\sigma} \uparrow \mu.$$

For the inductive hypothesis, $w_\sigma \in \omega_{L_a^\delta}$.

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

$$\sim a \in \text{OpFalse}_{IFP^{w_\sigma \uparrow (\alpha-1)}}^{w_\sigma} \downarrow \mu.$$

For the inductive hypothesis, $w_\sigma \in \omega_{M_a^\delta}$.

Consider a limit α . Then $K_a^\alpha = \bigcup_{\beta < \alpha} K_a^\beta$ and $K_{\sim a}^\alpha = \bigcup_{\beta < \alpha} K_{\sim a}^\beta$. If $a \in IFP^{w_\sigma \uparrow \alpha}$, then there exists a $\beta < \alpha$ such that $a \in IFP^{w_\sigma \uparrow \beta}$. For the inductive hypothesis, $w_\sigma \in \omega_{K_a^\beta}$ so $w_\sigma \in \omega_{K_a^\alpha}$. Similarly for $\sim a$. \square

We can now prove that $IFPP^{\mathcal{P}}$ is sound and complete.

Theorem 9 (Soundness and completeness of $IFPP^{\mathcal{P}}$). *For a ground probabilistic logic program \mathcal{P} , let K_a^α and $K_{\sim a}^\alpha$ be the formulas associated with atom a in $IFPP^{\mathcal{P}} \uparrow \alpha$. For every atom a and total choice σ , there is an iteration α_0 such that for all $\alpha > \alpha_0$ we have:*

$$w_\sigma \in \omega_{K_a^\alpha} \leftrightarrow WFM(w_\sigma) \models a \quad (3.4)$$

$$w_\sigma \in \omega_{K_{\sim a}^\alpha} \leftrightarrow WFM(w_\sigma) \models \sim a \quad (3.5)$$

Proof. The \rightarrow direction of Equations (3.4) and (3.5) is Lemma 8. In the other direction, $WFM(w_\sigma) \models a$ implies $\exists \alpha_0 \forall \alpha : \alpha \geq \alpha_0 \rightarrow IFP^{w_\sigma \uparrow \alpha} \models a$. For Lemma 9, $w_\sigma \in \omega_{K_a^\alpha}$. $WFM(w_\sigma) \models \sim a$ implies $\exists \alpha_0 \forall \alpha : \alpha \geq \alpha_0 \rightarrow IFP^{w_\sigma \uparrow \alpha} \models \sim a$. For Lemma 9, $w_\sigma \in \omega_{K_{\sim a}^\alpha}$. \square

We can now prove that every query for every sound program has a countable set of countable explanations that is covering.

Theorem 10 (Well-definedness of the distribution semantics). *For a sound ground probabilistic logic program \mathcal{P} , $\mu_{\mathcal{P}}(\{w | w \in W_{\mathcal{P}}, w \models a\})$ for all ground atoms a is well defined.*

Proof. Let K_a^δ and $K_{\sim a}^\delta$ be the formulas associated with atom a in $IFPP^{\mathcal{P}} \uparrow \delta$ where δ is the depth of the program. For the soundness and completeness of $IFPP^{\mathcal{P}}$, then $\{w | w \in W_{\mathcal{P}}, w \models a\} = \omega_{K_a^\delta}$.

Each iteration of $\text{OpTrue}_{IFPP^{\mathcal{P}} \uparrow \beta}^{\mathcal{P}}$ and $\text{OpFalse}_{IFPP^{\mathcal{P}} \uparrow \beta}^{\mathcal{P}}$ for all β generates countable sets of countable explanations since the set of rules is countable. So K_a^δ is a countable set of countable explanations and $\mu_{\mathcal{P}}(\{w | w \in W_{\mathcal{P}}, w \models a\})$ is well defined. \square

Moreover, if the program is sound, for all atoms a , $\omega_{K_a^\delta} = \omega_{K_{\sim a}^\delta}^c$ where δ is the depth of the program, as otherwise there would exist a world w_σ

such that $w_\sigma \notin \omega_{K_a^\delta}$ and $w_\sigma \notin \omega_{K_{\sim a}^\delta}$. But w_σ has a two-valued WFM so $WFM(w_\sigma) \models a$ or $WFM(w_\sigma) \models \sim a$. In the first case, $w_\sigma \in \omega_{K_a^\delta}$ and in the latter, $w_\sigma \in \omega_{K_{\sim a}^\delta}$ against the hypothesis.

To give a semantics to a program with function symbols in the other languages under the DS, we can translate it into ProbLog using the techniques from Section 2.4 and use the above semantics for ProbLog.

3.3 Comparison with Sato and Kameya's Definition

Sato and Kameya [2001] define the distribution semantics for definite programs, i.e., programs without negative literals. They build a probability measure on the set of Herbrand interpretations from a collection of finite distributions. Let the set of ground probabilistic facts \mathcal{F} be $\{f_1, f_2, \dots\}$ and let X_i be a random variable associated with f_i whose domain \mathcal{V}_i is $\{0, 1\}$.

They define the sample space $V_{\mathcal{F}}$ as a topological space with the product topology as the event space such that each $\{0, 1\}$ is equipped with the discrete topology.

In order to clarify this definition, let us introduce some topology terminology. A *topology* on a set V [Willard, 1970, page 23] is a collection Ψ of subsets of V , called the *open sets*, satisfying: (t-1) any union of elements of Ψ belongs to Ψ , (t-2) any finite intersection of elements of Ψ belongs to Ψ , (t-3) \emptyset and V belong to Ψ . We say that (V, Ψ) is a *topological space*. The *discrete topology* of a set V [Steen and Seebach, 2013, page 41] is the powerset $\mathbb{P}(V)$ of V .

The *infinite Cartesian product* of sets ψ_i for $i = 1, 2, \dots$ is

$$\rho = \prod_{i=1}^{\infty} \psi_i = \{(s_1, s_2, \dots) \mid s_i \in \psi_i, i = 1, 2, \dots\}$$

A *product topology* [Willard, 1970, page 53] on the infinite Cartesian product $\times_{i=1}^{\infty} \mathcal{V}_i$ is a set containing all possible unions of open sets of the form $\times_{i=1}^{\infty} \nu_i$ where (p-1) ν_i is open $\forall i$ and (p-2) for all but finitely many i , $\nu_i = \mathcal{V}_i$. Sets satisfying (p-1) and (p-2) are called *cylinder sets*. There exists a countable number of them.

So $V_{\mathcal{F}} = \times_{i=1}^{\infty} \{0, 1\}$, i.e., it is an infinite Cartesian product with $\mathcal{V}_i = \{0, 1\}$ for all i . Sato and Kameya [2001] define a probability measure $\eta_{\mathcal{F}}$ over the sample space $V_{\mathcal{F}}$ from a collection of finite joint distributions $P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n)$ for $n \geq 1$ such that

$$\left\{ \begin{array}{l} 0 \leq P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n) \leq 1 \\ \sum_{k_1, \dots, k_n} P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n) = 1 \\ \sum_{k_{n+1}} P_{\mathcal{F}}^{(n+1)}(X_1 = k_1, \dots, X_{n+1} = k_{n+1}) = P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n) \end{array} \right. \quad (3.6)$$

The last equation is called the *consistency condition* or *compatibility condition*. The *Kolmogorov consistency theorem* [Chow and Teicher, 2012, page 194] states that, if the distributions $P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n)$ satisfy the compatibility condition, there exists a probability space $(V_{\mathcal{F}}, \Psi_{\mathcal{F}}, \eta_{\mathcal{F}})$ where $\eta_{\mathcal{F}}$ is a unique probability measure on $\Psi_{\mathcal{F}}$, the minimal σ -algebra containing open sets of $V_{\mathcal{F}}$ such that for any n ,

$$\eta_{\mathcal{F}}(X_1 = k_1, \dots, X_n = k_n) = P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n). \quad (3.7)$$

$P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n)$ is defined by Sato and Kameya [2001] as

$$P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n) = \pi_1 \dots \pi_n$$

where $\pi_i = \Pi_i$ if $k_i = 1$ and $\pi_i = 1 - \Pi_i$ if $k_i = 0$, with Π_i the annotation of fact f_i . This definition clearly satisfies the properties in Equation (3.6).

The distribution $P_{\mathcal{F}}^{(n)}(X_1 = k_1, \dots, X_n = k_n)$ is then extended to a probability measure over the set of Herbrand interpretations of the whole program. Let $\mathcal{B}_{\mathcal{P}}$ be $\{a_1, a_2, \dots\}$ and let Y_i be a random variable associated with a_i whose domain is $\{0, 1\}$. Moreover, let $a^k = a$ if $k = 1$ and $a^k = \sim a$ if $k = 0$. $V_{\mathcal{P}}$ is the infinite Cartesian product $V_{\mathcal{P}} = \times_{i=1}^{\infty} \{0, 1\}$.

Measure $\eta_{\mathcal{F}}$ is extended to $\eta_{\mathcal{P}}$ by introducing a series of finite joint distributions $P_{\mathcal{P}}^{(n)}(Y_1 = k_1, \dots, Y_n = k_n)$ for $n = 1, 2, \dots$ by

$$[a_1^{k_1} \wedge \dots \wedge a_n^{k_n}]_{\mathcal{F}} = \{v \in V_{\mathcal{F}} \mid \text{lh}(v) \models a_1^{k_1} \wedge \dots \wedge a_n^{k_n}\}$$

where $\text{lh}(v)$ is the least Herbrand model of $\mathcal{R} \cup \mathcal{F}_v$, with $\mathcal{F}_v = \{f_i \mid v_i = 1\}$.

Then let

$$P_{\mathcal{P}}^{(n)}(Y_1 = k_1, \dots, Y_n = k_n) = \eta_{\mathcal{F}}([a_1^{k_1} \wedge \dots \wedge a_n^{k_n}]_{\mathcal{F}})$$

Sato and Kameya state that $[a_1^{k_1} \wedge \dots \wedge a_n^{k_n}]_{\mathcal{F}}$ is $\eta_{\mathcal{F}}$ -measurable and that, by definition, $P_{\mathcal{P}}^{(n)}$ satisfy the compatibility condition

$$\sum_{k_{n+1}} P_{\mathcal{P}}^{(n+1)}(Y_1 = k_1, \dots, Y_{n+1} = k_{n+1}) = P_{\mathcal{P}}^{(n)}(Y_1 = k_1, \dots, Y_n = k_n)$$

Hence, there exists a unique probability measure $\eta_{\mathcal{P}}$ over $\Psi_{\mathcal{P}}$ which is an extension of $\eta_{\mathcal{F}}$.

In order to relate this definition to the one of Section 3.2, we need to introduce some more terminology on σ -algebras.

Definition 26 (Infinite-dimensional product σ -algebra and space). *For any measurable spaces (W_i, Ω_i) , $i = 1, 2, \dots$, define*

$$\mathcal{G} = \bigcup_{m=1}^{\infty} \left\{ \bigtimes_{i=1}^{\infty} \omega_i \mid \omega_i \in \Omega_i, 1 \leq i \leq m \text{ and } \omega_i = W_i, i > m \right\}$$

$$\bigotimes_{i=1}^{\infty} \Omega_i = \sigma(\mathcal{G})$$

$$\bigtimes_{i=1}^{\infty} (W_i, \Omega_i) = \left(\bigtimes_{i=1}^{\infty} W_i, \bigotimes_{i=1}^{\infty} \Omega_i \right)$$

Then $\bigtimes_{i=1}^{\infty} (W_i, \Omega_i)$ is the infinite-dimensional product space and $\bigotimes_{i=1}^{\infty} \Omega_i$ is the infinite-dimensional product σ -algebra. This definition generalizes Definition 14 for the case of infinite dimensions.

It is clear that if $W_i = \{0, 1\}$ and $\Omega_i = \mathbb{P}(\{0, 1\})$ for all i , then \mathcal{G} is the set of all possible unions of cylinder sets, so it is the product topology on $\bigtimes_{i=1}^{\infty} W_i$ and

$$\bigtimes_{i=1}^{\infty} (W_i, \Omega_i) = (V_{\mathcal{F}}, \Psi_{\mathcal{F}})$$

i.e., the infinite-dimensional product space and the space composed by $V_{\mathcal{F}}$ and the minimal σ -algebra containing open sets of $V_{\mathcal{F}}$ coincide. Moreover, according to [Chow and Teicher, 2012, Exercise 1.3.6], $\Psi_{\mathcal{F}}$ is the minimal σ -algebra generated by cylinder sets.

An infinite Cartesian product $\rho = \bigtimes_{i=1}^{\infty} \nu_i$ is *consistent* if it is different from the empty set, i.e., if $\nu_i \neq \emptyset$ for all $i = 1, 2, \dots$. We can establish a bijective map $\gamma_{\mathcal{F}}$ between infinite consistent Cartesian products $\rho = \bigtimes_{i=1}^{\infty} \nu_i$ and composite choices: $\gamma_{\mathcal{F}}(\bigtimes_{i=1}^{\infty} \nu_i) = \{(f_i, \emptyset, \mathbf{k}_i) \mid \nu_i = \{\mathbf{k}_i\}\}$.

Lemma 10 (Elements of $\Psi_{\mathcal{F}}$ as Countable Unions). *Each element of $\Psi_{\mathcal{F}}$ can be written as a countable union of consistent possibly infinite Cartesian products:*

$$\psi = \bigcup_{j=1}^{\infty} \rho_j \tag{3.8}$$

where $\rho_j = \bigtimes_{i=1}^{\infty} \nu_i$ and $\nu_i \in \{\{0\}, \{1\}, \{0, 1\}\}$ for all $i = 1, 2, \dots$

Proof. We will show that $\Psi_{\mathcal{F}}$ and Φ , the set of all elements of the form of Equation (3.8), coincide. Given a ψ for the form of Equation (3.8), each ρ_j can be written as a countable union of cylinder sets, so it belongs to $\Psi_{\mathcal{F}}$. Since $\Psi_{\mathcal{F}}$ is a σ -algebra and ψ is a countable union, then $\psi \in \Psi_{\mathcal{F}}$ and $\Phi \subseteq \Psi_{\mathcal{F}}$.

We can prove that Φ is a σ -algebra using the same technique of Lemma 2, where Cartesian products replace composite choices. Φ contains cylinder sets: even if each ρ_j must be consistent, inconsistent sets are empty set of worlds, so they can be removed from the union in Equation (3.8). As $\Psi_{\mathcal{F}}$ is the minimal σ -algebra containing cylinder sets, then $\Psi_{\mathcal{F}} \subseteq \Phi$. \square

So each element ψ of $\Psi_{\mathcal{F}}$ can be written as a countable union of consistent possibly infinite Cartesian products.

Lemma 11 ($\Gamma_{\mathcal{F}}$ is Bijective). *Consider the function $\Gamma_{\mathcal{F}} : \Psi_{\mathcal{F}} \rightarrow \Omega_{\mathcal{P}}$ defined by $\Gamma_{\mathcal{F}}(\psi) = \omega_K$ where $\psi = \bigcup_{j=1}^{\infty} \rho_j$ and $K = \bigcup_{j=1}^{\infty} \{\gamma_{\mathcal{F}}(\rho_j)\}$. Then $\Gamma_{\mathcal{F}}$ is bijective.*

Proof. Immediate because $\gamma_{\mathcal{F}}$ is bijective. \square

Theorem 11 (Equivalence with Sato and Kameya's definition). *Probability measure $\mu_{\mathcal{P}}$ coincides with $\eta_{\mathcal{P}}$ for definite programs.*

Proof. Consider $(X_1 = k_1, \dots, X_n = k_n)$ and let K be

$$\{(f_1, \emptyset, k_1), \dots, (f_n, \emptyset, k_n)\}.$$

Then $K = \Gamma_{\mathcal{F}}(\{(k_1, \dots, k_n, v_{n+1}, \dots) | v_i \in \{0, 1\}, i = n + 1, \dots\})$ and $\mu_{\mathcal{P}}$ assigns probability $\pi_1 \dots \pi_n$ to K , where $\pi_i = \Pi_i$ if $k_i = 1$ and $\pi_i = 1 - \Pi_i$ otherwise.

So $\mu_{\mathcal{P}}$ is in accordance with $P_{\mathcal{F}}^{(n)}$. But $P_{\mathcal{F}}^{(n)}$ can be extended in only one way to a probability measure $\eta_{\mathcal{F}}$ and there is a bijection between $\Psi_{\mathcal{F}}$ and $\Omega_{\mathcal{P}}$, so $\mu_{\mathcal{P}}$ is in accordance with $\eta_{\mathcal{F}}$ on all $\Psi_{\mathcal{F}}$.

Now consider $C = a_1^{k_1} \wedge \dots \wedge a_n^{k_n}$. Since $IFPP^{\mathcal{P}} \uparrow \delta$ is such that K_a^{δ} and $K_{\sim a}^{\delta}$ are countable sets of countable composite choices for all atoms a , we can compute a covering set of explanations K for C by taking a finite conjunction of countable sets of composite choices, so K is a countable set of countable composite choices.

Clearly, $P_{\mathcal{P}}^{(n)}(Y_1 = k_1, \dots, Y_n = k_n)$ coincides with $\mu_{\mathcal{P}}(\omega_K)$. But $P_{\mathcal{P}}^{(n)}$ can be extended in only one way to a probability measure $\eta_{\mathcal{P}}$, so $\mu_{\mathcal{P}}$ is in accordance with $\eta_{\mathcal{P}}$ on all $\Psi_{\mathcal{P}}$ when \mathcal{P} is a definite program. \square

4

Semantics for Hybrid Programs

The languages presented in Chapter 2 allow the definition of discrete random variables only. However, some domains naturally include continuous random variables. Probabilistic logic programs including continuous variables are called *hybrid*.

In this chapter, we discuss some languages for hybrid programs together with their semantics.

4.1 Hybrid ProbLog

Hybrid ProbLog [Gutmann et al., 2011a] extends ProbLog with continuous probabilistic facts of the form

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

where X is a logical variable appearing in the atom f and ϕ is an atom specifying a continuous distribution, such as, for example, *gaussian*(0, 1) to indicate a Gaussian distribution with mean 0 and standard deviation 1. Variables X of this form are called *continuous variables*.

A Hybrid ProbLog program \mathcal{P} is composed of definite rules \mathcal{R} and facts $\mathcal{F} = \mathcal{F}^d \cup \mathcal{F}^c$ where \mathcal{F}^d are discrete probabilistic facts as in ProbLog and \mathcal{F}^c are continuous probabilistic facts.

Example 48 (Gaussian mixture – Hybrid ProbLog). *A Gaussian mixture model is a way to generate values of a continuous random variable: a discrete random variable is sampled and, depending on the sampled value, a different Gaussian distribution is selected for sampling the value of the continuous variable.*

A Gaussian mixture model with two components can be expressed in Hybrid ProbLog as [Gutmann et al., 2011a]:

$0.6 :: \text{heads}.$
 $\text{tails} \leftarrow \sim \text{heads}.$
 $(X, \text{gaussian}(0, 1)) :: g(X).$
 $(X, \text{gaussian}(5, 2)) :: h(X).$
 $\text{mix}(X) \leftarrow \text{heads}, g(X).$
 $\text{mix}(X) \leftarrow \text{tails}, h(X).$
 $\text{pos} \leftarrow \text{mix}(X), \text{above}(X, 0).$

where, for example, $(X, \text{gaussian}(0, 1)) :: g(X)$ is a continuous probabilistic fact stating that X follows a Gaussian with mean 0 and standard deviation 1. The values of X in $\text{mix}(X)$ are distributed according to a mixture of two Gaussians. The atom pos is true if X is positive.

A number of predicates are defined for manipulating a continuous variable X :

- $\text{below}(X, c)$ succeeds if $X < c$ where c is a numeric constant;
- $\text{above}(X, c)$ succeeds if $X > c$ where c is a numeric constant;
- $\text{ininterval}(X, c_1, c_2)$ succeeds if $X \in [c_1, c_2]$ where c_1 and c_2 are numeric constants.

Continuous variables can't be unified with terms or used in Prolog comparison and arithmetic operators, so in the example above, it is not possible to use expressions $g(0)$, $(g(X), X > 0)$, $(g(X), 3 * X + 4 > 4)$ in the body of rules. The first two, however, can be expressed as $g(X), \text{ininterval}(X, 0, 0)$ and $g(X), \text{above}(X, 0)$, respectively.

Hybrid ProbLog assumes a finite set of continuous probabilistic facts and no function symbols, so the set of continuous variables is finite. Let us indicate the set with $\mathbf{X} = \{X_1, \dots, X_n\}$, defined by the set of atoms for probabilistic facts $F = \{f_1, \dots, f_n\}$ where each f_i is ground except for variable X_i . An assignment $\mathbf{x} = \{x_1, \dots, x_n\}$ to \mathbf{X} defines a substitution $\theta_{\mathbf{x}} = \{X_1/x_1, \dots, X_n/x_n\}$ and, in turn, a set of ground facts $F\theta_{\mathbf{x}}$.

Given a selection σ for discrete facts and an assignment \mathbf{x} to continuous variables, a world $w_{\sigma, \mathbf{x}}$ is defined as

$$w_{\sigma, \mathbf{x}} = \mathcal{R} \cup \{f\theta \mid (f, \theta, 1) \in \sigma\} \cup F\theta_{\mathbf{x}}$$

Each continuous variable X_i is associated with a probability density $p_i(X_i)$. Since all the variables are independent, $p(\mathbf{x}) = \prod_{i=1}^n p_i(x_i)$ is a joint probability density over \mathbf{X} . $p(\mathbf{X})$ and $P(\sigma)$ then define a joint probability density over the worlds:

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

The probability of a ground atom q different from the atom of a continuous probabilistic fact is then defined as in the DS for discrete programs as

$$\begin{aligned} P(q) &= \int_{\sigma \in S_{\mathcal{P}}, \mathbf{x} \in \mathbb{R}^n} p(q, w_{\sigma, \mathbf{x}}) = \\ &\int_{\sigma \in S_{\mathcal{P}}, \mathbf{x} \in \mathbb{R}^n} P(q|w_{\sigma, \mathbf{x}})p(w_{\sigma, \mathbf{x}}) = \\ &\int_{\sigma \in S_{\mathcal{P}}, \mathbf{x} \in \mathbb{R}^n : w_{\sigma, \mathbf{x}} \models q} p(w_{\sigma, \mathbf{x}}) \end{aligned}$$

where $S_{\mathcal{P}}$ is the set of all selections over discrete probabilistic facts. If the set $\{(\sigma, \mathbf{x}) | \sigma \in S_{\mathcal{P}}, \mathbf{x} \in \mathbb{R}^n : w_{\sigma, \mathbf{x}} \models q\}$ is measurable, then the probability is well defined. Gutmann et al. [2011a] prove that, for each instance σ , the set $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^n : w_{\sigma, \mathbf{x}} \models q\}$ is an n -dimensional interval $I = [a_1, b_1] \times \dots \times [a_n, b_n]$ of \mathbb{R}^n where a_i and b_i can be $-\infty$ and $+\infty$, respectively, for $i = 1, \dots, n$. The probability that $\mathbf{X} \in I$ is then given by

$$P(\mathbf{X} \in I) = \int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} p(\mathbf{x}) d\mathbf{x} \quad (4.1)$$

The proof is based on considering a *discretized theory* \mathcal{P}_D obtained as

$$\mathcal{P}_D = \mathcal{R} \cup \mathcal{F}^d \cup \{\text{below}(X, C), \text{above}(X, C), \text{ininterval}(X, C1, C2)\} \cup \{f\{X/f\} | (X, \phi) :: f \in \mathcal{F}^d\}$$

The discretized program is a regular ProbLog program, so we can consider its worlds. In each world, we can derive the query by SLD resolution and keep track of the instantiations of the facts for the comparison predicates used in each proof. This yields a set of comparison facts that defines an n -dimensional interval. So we can compute the probability of the query in the world with Equation (4.1). By summing these values for all proofs, we get the probability of the query in the world of the ProbLog program. The weighted sum of the probabilities in the worlds over all worlds of \mathcal{P}_D gives the probability of the query, where the weights are the probabilities of the worlds according to the discrete facts.

Example 49 (Query over a Gaussian mixture – Hybrid ProbLog). *For Example 48, the discretized program is*

$0.6 :: \text{heads}.$
 $\text{tails} \leftarrow \sim\text{heads}.$
 $g(g(X)).$
 $h(h(X)).$
 $\text{mix}(X) \leftarrow \text{heads}, g(X).$
 $\text{mix}(X) \leftarrow \text{tails}, h(X).$
 $\text{pos} \leftarrow \text{mix}(X), \text{above}(X, 0).$
 $\text{below}(X, C).$
 $\text{above}(X, C).$
 $\text{ininterval}(X, C1, C2).$

which is a ProbLog program with two worlds. In the one containing heads, the only proof for the query pos uses the fact $\text{above}(g(g(X)), 0)$, so the probability of pos in this world is

$$P(\text{pos}|\text{heads}) = \int_0^{\infty} p(x)dx = 1 - F(0, 0, 1)$$

where $F(x, \mu, \sigma)$ is the Gaussian cumulative distribution with mean μ and standard deviation σ . Therefore, $P(\text{pos}|\text{heads}) = 0.5$.

In the world not containing heads, the only proof for the query pos uses the fact $\text{above}(h(h(X)), 0)$, so the probability of pos in this world is $P(\text{pos}|\sim\text{heads}) = 1 - F(0, 5, 2) \approx 0.994$. So overall

$$P(\text{pos}) = 0.6 \cdot 0.5 + 0.4 \cdot 0.994 \approx 0.698$$

This approach for defining the semantics also defines a strategy for performing inference. However, Hybrid ProbLog imposes severe restrictions on the programs, not allowing the use of continuous variables in expressions possibly involving other variables.

4.2 Distributional Clauses

Distributional Clauses (DCs) [Gutmann et al., 2011c] are definite clause with an atom $h \sim \mathcal{D}$ in the head, where h is a term, \sim is a binary predicate used in infix notation, and \mathcal{D} is a term specifying a discrete or continuous probability distribution. For each ground instance $(h \sim \mathcal{D} \leftarrow b_1, \dots, b_n)\theta$, where θ is a substitution over the Herbrand universe of the logic program, a distributional clause defines a random variable $h\theta$ with the distribution indicated by $\mathcal{D}\theta$ whenever all the $b_i\theta$ hold. The term \mathcal{D} can be non-ground, i.e., distribution parameters can be related to conditions in the body, similarly to flexible probabilities, see Section 2.7.

The reserved functor $\simeq/1$ is used to represent the outcome of a random variable: $\simeq d$, for example, indicates the outcome of random variable d . The set of special predicates

$$dist_rel = \{dist_eq/2, dist_lt/2, dist_leq/2, dist_gt/2, dist_geq/2\}$$

can be used to compare the outcome of a random variable with a constant or the outcome of another random variable. These predicates are assumed to be defined by ground facts, one for each true atom for one of the predicates. Terms of the form $\simeq(h)$ can also be used in other predicates with some restrictions, e.g., it does not make sense to unify the outcome of a continuous random variable with a constant or another random variable as the probability that they are equal has measure 0. The predicate $\sim=/2$ is used to unify discrete random variables with terms: $h\sim=v$ means $\simeq(h) = v$, which is true iff the value of the random variable h unifies with v .

A DC programs \mathcal{P} is composed by a set of definite clauses \mathcal{R} and a set of DCs \mathcal{C} . A *world* of \mathcal{P} is the program $\mathcal{R} \cup \mathcal{F}$ where \mathcal{F} is the set of ground atoms for the predicates in $dist_rel$ that are true for each random variable $h\theta$ defined by the program.

Let us now see two examples.

Example 50 (Gaussian mixture – DCs). *The Gaussian mixture model of Example 48 can be expressed with DCs as*

$$coin \sim [0.6 : heads, 0.4 : tails].$$

$$g \sim gaussian(0, 1).$$

$$h \sim gaussian(5, 2).$$

$$mix(X) \leftarrow dist_eq(\simeq coin, heads), g\sim=X.$$

$$mix(X) \leftarrow dist_eq(\simeq coin, tails), h\sim=X.$$

$$pos \leftarrow mix(X), dist_gt(X, 0).$$

where, for example, g follows a Gaussian with mean 0 and standard deviation 1.

This example shows that Hybrid ProbLog programs can be expressed in DCs.

Example 51 (Moving people – DCs [Nitti et al., 2016]). *The following program models a set of people moving on a real line:*

$$n \sim poisson(6).$$

$$pos(P) \sim uniform(0, M) \leftarrow n\sim=N, between(1, N, P),$$

$$M \text{ is } 10 * N.$$

$$left(A, B) \leftarrow dist_lt(\simeq pos(A), \simeq pos(B)).$$

where $\text{between}(1, N, P)$ is a predicate that, when P is not bound, generates all integers between 1 and N , and $\text{is}/2$ is the standard Prolog predicate for the evaluation of expressions.

The first clause defines the number of people n as a random variable that follows a Poisson distribution with mean 6. The position of each person is modeled with the random variable $\text{pos}(P)$ defined in the second clause as a continuous random variable uniformly distributed from 0 to $M = 10n$ (that is, 10 times the number of people), for each integer person identifier P such that $1 \leq P \leq n$. If the value of n is 2, for example, there will be two independent random variables $\text{pos}(1)$ and $\text{pos}(2)$ with distribution $\text{uniform}(0, 20)$. The last clause defines relation $\text{left}/2$ between people positions.

A DC program must satisfy some conditions to be considered valid.

Definition 27 (Valid program [Gutmann et al., 2011c]). A DC program \mathcal{P} is called valid if the following conditions are fulfilled:

- (V1) In the relation $h \sim \mathcal{D}$ that holds in the least fixpoint of a program, there is a functional dependency from h to \mathcal{D} , so there is a unique ground distribution \mathcal{D} for each ground random variable h .
- (V2) The program is distribution-stratified, that is, there exists a function $\text{rank}(\cdot)$ that maps ground atoms to \mathbb{N} and satisfies the following properties: (1) for each ground instance of a clause $h \sim \mathcal{D} \leftarrow b_1, \dots, b_n$, it holds that $\text{rank}(h \sim \mathcal{D}) > \text{rank}(b_i)$ for all i ; (2) for each ground instance of a regular program clause $h \leftarrow b_1, \dots, b_n$, it holds $\text{rank}(h) \geq \text{rank}(b_i)$ for all i ; and (3) for each ground atom b that contains (the name of) a random variable h , $\text{rank}(b) \geq \text{rank}(h \sim \mathcal{D})$ (with $h \sim \mathcal{D}$ being the head of the distribution clause defining h). The ranking is extended to random variables by setting $\text{rank}(h) = \text{rank}(h \sim \mathcal{D})$.
- (V3) The indicator functions (see below) for all ground probabilistic facts are Lebesgue-measurable.
- (V4) Each atom in the least fixpoint can be derived from a finite number of probabilistic facts (finite support condition [Sato, 1995]).

We now define the series of distributions $P_{\mathcal{F}}^{(n)}$ as in Sato and Kameya's definition of the DS of Equation (3.6). We consider an enumeration $\{f_1, f_2, \dots\}$ of atoms for the predicates in dist_rel such that $i < j \Rightarrow \text{rank}(f_i) \leq \text{rank}(f_j)$ where $\text{rank}(\cdot)$ is a ranking function as in Definition 27. We also define, for each predicate $\text{rel}/2 \in \text{dist_rel}$, an indicator function:

$$I_{rel}^1(X_1, X_2) = \begin{cases} 1 & \text{if } rel(X_1, X_2) \text{ is true,} \\ 0 & \text{if } rel(X_1, X_2) \text{ is false} \end{cases}$$

$$I_{rel}^0(X_1, X_2) = 1.0 - I_{rel}^1(X_1, X_2)$$

We define the probability distributions $P_{\mathcal{F}}^{(n)}$ over finite sets of ground facts f_1, \dots, f_n using expectations of indicator functions. Let $x_i \in \{1, 0\}$ for $i = 1, \dots, n$ be truth values for the facts f_1, \dots, f_n . Let $\{rv_1, \dots, rv_m\}$ be the set of random variables these n facts depend upon, ordered such that if $rank(rv_i) < rank(rv_j)$, then $i < j$, and let $f_i = rel_i(t_{i1}, t_{i2})$. Let $\theta^{-1} = \{(rv_1)/V_1, \dots, (rv_m)/V_m\}$ be an antisubstitution (see Section 1.3) that replaces evaluations of the random variables with real variables for integration. Then the probability distributions $P_{\mathcal{F}}^{(n)}$ are given by

$$P_{\mathcal{F}}^{(n)}(f_1 = x_1, \dots, f_n = x_n) =$$

$$E[I_{rel_1}^{x_1}(t_{11}, t_{12}), \dots, I_{rel_n}^{x_n}(t_{n1}, t_{n2})] =$$

$$\int \dots \int I_{rel_1}^{x_1}(t_{11}\theta^{-1}, t_{12}\theta^{-1}) \dots I_{rel_n}^{x_n}(t_{n1}\theta^{-1}, t_{n2}\theta^{-1})$$

$$d\mathcal{D}_{rv_1}(V_1) \dots d\mathcal{D}_{rv_m}(V_m) \quad (4.2)$$

For valid DC programs, the following proposition can be proved.

Proposition 4 (Valid DC Program [Gutmann et al., 2011c]). *Let \mathcal{P} be a valid DC program. \mathcal{P} defines a probability measure $P_{\mathcal{P}}$ over the set of fixpoints of operator T_w where w is a world of \mathcal{P} . Hence, for an arbitrary formula q over atoms, \mathcal{P} also defines the probability that q is true.*

The semantics of DCs coincides with that of hybrid ProbLog on programs that satisfy the constraints imposed by the latter.

An alternative view of the semantics can be given by means of a stochastic $T_{\mathcal{P}}$ operator, $ST_{\mathcal{P}}$, extending the $T_{\mathcal{P}}$ operator of Definition 1 to deal with probabilistic facts $dist_rel(t_1, t_2)$. We need a function $READTABLE(\cdot)$ that evaluates probabilistic facts and stores the sampled values for random variables. $READTABLE$ applied to probabilistic fact $dist_rel(t_1, t_2)$ returns the truth value of the fact evaluated on the basis of the values of the random variables in the arguments. The values are either retrieved from the table or sampled according to their distribution the first time they are accessed. In the case they are sampled, they are stored for future use.

Definition 28 ($ST_{\mathcal{P}}$ operator [Gutmann et al., 2011c]). *Let \mathcal{P} be a valid DC program. Starting from a set of ground facts I , the $ST_{\mathcal{P}}$ operator is defined as*

$$ST_{\mathcal{P}}(I) = \{h \mid h \leftarrow b_1 \dots, b_n \in \text{ground}(\mathcal{P}) \wedge \forall b_i : b_i \in I \vee \quad (4.3)$$

$$b_i = \text{dist_rel}(t_1, t_2) \wedge$$

$$(t_j = \simeq h \Rightarrow h \sim \mathcal{D} \in I \wedge \text{READTABLE}(b_i) = \text{true})\}$$

Computing the least fixpoint of the $ST_{\mathcal{P}}$ operator returns a possible model of the program. The $ST_{\mathcal{P}}$ operator is stochastic, so it defines a sampling process. The distribution over models defined by $ST_{\mathcal{P}}$ is the same as that defined by Equation (4.2).

Example 52 ($ST_{\mathcal{P}}$ for moving people – DC [Nitti et al., 2016]). *Given the DC program \mathcal{P} defined in Example 51, a possible sequence of application of the $ST_{\mathcal{P}}$ operator is*

$ST_{\mathcal{P}} \uparrow \alpha$	Table
\emptyset	\emptyset
$\{n \sim \text{poisson}(6)\}$	$\{n = 2\}$
$\{n \sim \text{poisson}(6), \text{pos}(1) \sim \text{uniform}(0, 20), \text{pos}(2) \sim \text{uniform}(0, 20)\}$	$\{n = 2, \text{pos}(1) = 3.1, \text{pos}(2) = 4.5\}$
$\{n \sim \text{poisson}(6), \text{pos}(1) \sim \text{uniform}(0, 20), \text{pos}(2) \sim \text{uniform}(0, 20), \text{left}(1, 2)\}$	$\{n = 2, \text{pos}(1) = 3.1, \text{pos}(2) = 4.5\}$

Nitti et al. [2016] proposed a modification of DCs where the relation symbols are replaced with their Prolog versions, the clauses can contain negation, and the $ST_{\mathcal{P}}$ operator is defined slightly differently.

The new version of the $ST_{\mathcal{P}}$ operator does not use a separate table for storing the values sampled for random variables and does not store distribution atoms.

Definition 29 ($ST_{\mathcal{P}}$ operator [Nitti et al., 2016]). *Let \mathcal{P} be a valid DC program. Starting from a set of ground facts I , the $ST_{\mathcal{P}}$ operator is defined as*

$$ST_{\mathcal{P}}(I) = \{h = v \mid h \sim \mathcal{D} \leftarrow b_1, \dots, b_n \in \text{ground}(\mathcal{P}) \wedge \forall b_i :$$

$$b_i \in I \vee b_i = \text{dist_rel}(t_1, t_2) \wedge t_1 = v_1 \in I \wedge t_2 = v_2 \in I \wedge$$

$$\text{dist_rel}(v_1, v_2) \wedge v \text{ is sampled from } \mathcal{D}\} \cup$$

$$\{h \mid h \leftarrow b_1, \dots, b_n \in \text{ground}(\mathcal{P}) \wedge h \neq (r \sim \mathcal{D}) \wedge \forall b_i :$$

$$b_i \in I \vee b_i = \text{dist_rel}(t_1, t_2) \wedge t_1 = v_1 \in I \wedge$$

$$t_2 = v_2 \in I \wedge \text{dist_rel}(v_1, v_2)\}$$

where *dist_rel* is one of $=, <, \leq, >, \geq$. In practice, for each distributional clause $h \sim \mathcal{D} \leftarrow b_1 \dots, b_n$, whenever the body $b_1 \dots, b_n$ is true in I , a value v for random variable h is sampled from the distribution \mathcal{D} and $h = v$ is added to the interpretation. Similarly for deterministic clauses, adding ground atoms whenever the body is true.

We can provide an alternative definition of world. A world is obtained in a number of steps. First, we must distinguish the logical variables that can hold continuous values from those that can hold values from the logical Herbrand universe. We replace the discrete logical variables with terms from the Herbrand universe in all possible ways. Then we sample, for each distributional clause $h \sim \mathcal{D} \leftarrow b_1 \dots, b_n$ in the resulting program, a value for h from distribution \mathcal{D} and replace the clause with $h \sim v \leftarrow b_1 \dots, b_n$.

If we compute the least fixpoint of T_w for a world w sampled in this way, we get a model that is the same as the one that is the least fixpoint of $ST_{\mathcal{P}}$ if all random variables are sampled in the same way. So the least fixpoint of $ST_{\mathcal{P}}$ is a model of at least one world. The probability measure over models defined by $ST_{\mathcal{P}}$ of Definition 30 coincides with that defined by $ST_{\mathcal{P}}$ of Definition 28 and Equation (4.2). Nitti et al. [2016] call worlds the possible models but we prefer to use the word world for sampled normal programs.

Example 53 ($ST_{\mathcal{P}}$ for moving people – DC [Nitti et al., 2016]).

Given the DC program \mathcal{P} defined in Example 51 and the sequence of applications of the $ST_{\mathcal{P}}$ operator of Definition 28, the corresponding sequence of applications for $ST_{\mathcal{P}}$ of Definition 30 is

$$\begin{aligned} ST_{\mathcal{P}} \uparrow 0 &= \emptyset \\ ST_{\mathcal{P}} \uparrow 1 &= \{n = 2\} \\ ST_{\mathcal{P}} \uparrow 2 &= \{n = 2, pos(1) = 3.1, pos(2) = 4.5\} \\ ST_{\mathcal{P}} \uparrow 3 &= \{n = 2, pos(1) = 3.1, pos(2) = 4.5, left(1, 2)\} \\ ST_{\mathcal{P}} \uparrow 4 &= ST_{\mathcal{P}} \uparrow 3 = \text{lfp}(ST_{\mathcal{P}}) \end{aligned}$$

so $\{n = 2, pos(1) = 3.1, pos(2) = 4.5, left(1, 2)\}$ is a possible model or a model of a world of the program.

Regarding negation, since programs need to be stratified to be valid, negation poses no particular problem: the $ST_{\mathcal{P}}$ operator is applied at each rank from lowest to highest, along the same lines as the perfect model semantics [Przymusiński, 1988].

Given a DC program \mathcal{P} and a negative literal $l = \sim a$, to determine its truth in \mathcal{P} , we can proceed as follows. Consider the interpretation I obtained

by applying the $ST_{\mathcal{P}}$ operator until the least fixpoint for $rank(l)$ is reached (or exceeded). If a is a non-comparison atomic formula, I is true if $a \notin I$ and false otherwise. If a is a comparison atom involving a random variable r such as $l = (r \sim val)$, then l is true whenever $\exists val' : r = val' \in I$ with $val \neq val'$ or $\nexists val' : r = val' \in I$, i.e., r is not defined in I . Note that I is the least fixpoint for $rank(r \sim \mathcal{D})$ (or higher), thus $\nexists val' : r = val' \in I$ implies that r is not defined also in the following applications of the $ST_{\mathcal{P}}$ operator and so also in the possible model.

Example 54 (Negation in DCs [Nitti et al., 2016]). *Consider an example where we draw $nballs$ balls with replacement from an urn containing an equal number of red, blue, and black balls. $nballs$ follows a Poisson distribution and the color of each ball is a random variable uniformly distributed over the set $\{red, blue, black\}$:*

$nballs \sim poisson(6)$.

$color(X) \sim uniform([red, blue, black]) \leftarrow$

$nballs \simeq N, between(1, N, X)$.

$not_red \leftarrow \sim color(2) \simeq red$.

not_red is true in those possible models where the value of $color(2)$ is not red or in those in which $color(2)$ is not defined, for example, when $n = 1$.

4.3 Extended PRISM

Islam et al. [2012b] proposed an extension of PRISM that includes continuous random variables with a Gaussian or gamma distribution.

The set_sw directives allow the definition of probability density functions. For instance, $set_sw(r, norm(Mu, Var))$ specifies that the outcomes of random processes r have Gaussian distribution with mean Mu and variance Var .

Parameterized families of random processes may be specified, as long as the parameters are discrete-valued. For instance,

$$set_sw(w(M), norm(Mu, Var))$$

specifies a family of random variables, with one for each value of M . As in PRISM, the distribution parameters may be computed as functions of M .

Moreover, PRISM is extended with linear equality constraints over reals. Without loss of generality, we assume that constraints are written as linear equalities of the form $Y = a_1 \cdot X_1 + \dots + a_n \cdot X_n + b$ where a_i and b are all floating point constants.

In the following, we use *Constr* to denote a set (conjunction) of linear equality constraints. We also denote by X a vector of variables and/or values, explicitly specifying the size only when it is not clear from the context. This allows us to write linear equality constraints compactly (e.g., $Y = a \cdot X + b$).

Example 55 (Gaussian mixture – Extended PRISM). *The Gaussian mixture model of Example 48 can be expressed with Extended PRISM as*

```

mix( $X$ )  $\leftarrow$  msw(coin, heads), msw( $g$ ,  $X$ ).
mix( $X$ )  $\leftarrow$  msw(coin, tails), msw( $h$ ,  $X$ )
values(coin, [heads, tails]).
values( $g$ , real).
values( $h$ , real),
 $\leftarrow$  set_sw(coin, [0.6, 0.4]).
 $\leftarrow$  set_sw( $g$ , norm(0, 1)).
 $\leftarrow$  set_sw( $h$ , norm(5, 2)).

```

Let us now show an example with constraints.

Example 56 (Gaussian mixture and constraints – Extended PRISM). *Consider a factory with two machines a and b . Each machine produces a widget with a continuous feature. A widget is produced by machine a with probability 0.3 and by machine b with probability 0.7. If the widget is produced by machine a , the feature is distributed as a Gaussian with mean 2.0 and variance 1.0. If the widget is produced by machine b , the feature is distributed as a Gaussian with mean 3.0 and variance 1.0. The widget then is processed by a third machine that adds a random quantity to the feature. The quantity is distributed as a Gaussian with mean 0.5 and variance 1.5. This is encoded by the program:*

```

widget( $X$ )  $\leftarrow$ 
  msw( $m$ ,  $M$ ), msw(st( $M$ ),  $Z$ ), msw(pt,  $Y$ ),  $X = Y + Z$ .
values( $m$ , [ $a$ ,  $b$ ]).
values(st(-), real).
values(pt, real).
 $\leftarrow$  set_sw( $m$ , [0.3, 0.7]).
 $\leftarrow$  set_sw(st( $a$ ), norm(2.0, 1.0)).
 $\leftarrow$  set_sw(st( $b$ ), norm(3.0, 1.0)).
 $\leftarrow$  set_sw(pt, norm(0.5, 0.1)).

```

The semantics extends the DS for the discrete case by defining a probability space for the *msw* switches and then extending it to a probability space for the entire program using the least model semantics of constraint logic programs [Jaffar et al., 1998].

The probability space for the probabilistic facts is constructed from those of discrete and continuous random variables. The probability space for N continuous random variables is the Borel σ -algebra over \mathbb{R}^N and a Lebesgue measure on this set is the probability measure. This is combined with the space for discrete random variables using the Cartesian product. The probability space for facts is extended to the space of the entire program using the least model semantics: a point in this space is an arbitrary interpretation of the program obtained from a point in the space of the facts by means of logical consequence. A probability measure over the space of the program is then defined by using the measure defined for the probabilistic facts alone. The semantics for programs without constraints is essentially equivalent to the one of DC.

The authors propose an exact inference algorithm that extends the one of PRISM by reasoning symbolically over the constraints on the random variables, see Section 5.11. This is permitted by the restrictions on the types of distributions, Gaussian and gamma, and on the form of constraints, linear equations.

4.4 cplint Hybrid Programs

`cplint` handles continuous random variables with its sampling inference module. The user can specify a probability density on an argument *Var* of an atom *a* with rules of the form

$$a : \textit{Density} \leftarrow \textit{Body}$$

where *Density* is a special atom identifying a probability density on variable *Var* and *Body* (optional) is a regular clause body. *Density* atoms can be:

- *uniform*(*Var*, *L*, *U*): *Var* is uniformly distributed in $[L, U]$.
- *gaussian*(*Var*, *Mean*, *Variance*): Gaussian distribution with parameters *Mean* and *Variance*. The distribution can be multivariate if *Mean* is a list and *Variance* a list of lists representing the mean vector and the covariance matrix, respectively. In this case, the values of *Var* are lists of real values with the same length as that of *Mean*.
- *dirichlet*(*Var*, *Par*): *Var* is a list of real numbers following a Dirichlet distribution with α parameters specified by the list *Par*.
- *gamma*(*Var*, *Shape*, *Scale*) gamma distribution with parameters *Shape* and *Scale*.

- $\text{beta}(\text{Var}, \text{Alpha}, \text{Beta})$ beta distribution with parameters Alpha and Beta .
- $\text{poisson}(\text{Var}, \text{Lambda})$ Poisson distribution with parameter Lambda .
- $\text{binomial}(\text{Var}, N, P)$ binomial distribution with parameters N and P .
- $\text{geometric}(\text{Var}, P)$ geometric distribution with parameter P .

For example

$$g(X) : \text{gaussian}(X, 0, 1).$$

states that argument X of $g(X)$ follows a Gaussian distribution with mean 0 and variance 1, while

$$g(X) : \text{gaussian}(X, [0, 0], [[1, 0], [0, 1]]).$$

states that argument X of $g(X)$ follows a Gaussian multivariate distribution with mean vector $[0, 0]$ and covariance matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Example 57 (Gaussian mixture – *cplint*). *Example 48 of a mixture of two Gaussians can be encoded as¹:*

heads : 0.6 ; *tails* : 0.4.
 $g(X) : \text{gaussian}(X, 0, 1).$
 $h(X) : \text{gaussian}(X, 5, 2).$
 $\text{mix}(X) \leftarrow \text{heads}, g(X).$
 $\text{mix}(X) \leftarrow \text{tails}, h(X).$

The argument X of $\text{mix}(X)$ follows a distribution that is a mixture of two Gaussian, one with mean 0 and variance 1 with probability 0.6 and one with mean 5 and variance 2 with probability 0.4.

The parameters of the distribution atoms can be taken from the probabilistic atom.

Example 58 (Estimation of the mean of a Gaussian – *cplint*). *The program²*

$\text{value}(I, X) \leftarrow \text{mean}(M), \text{value}(I, M, X).$
 $\text{mean}(M) : \text{gaussian}(M, 1.0, 5.0).$
 $\text{value}(_, M, X) : \text{gaussian}(X, M, 2.0).$

¹http://cplint.eu/e/gaussian_mixture.pl

²http://cplint.eu/e/gauss_mean_est.pl

states that, for an index I , the continuous variable X is sampled from a Gaussian whose variance is 2 and whose mean M is sampled from a Gaussian with mean 1 and variance 5.

This program can be used to estimate the mean of a Gaussian by querying $\text{mean}(M)$ given observations for atom $\text{value}(I, X)$ for different values of I .

Any operation is allowed on continuous random variables.

Example 59 (Kalman filter – `cplint`). A Kalman filter [Harvey, 1990] is a dynamical system, i.e., a system that evolves with time. At every integer time point t , the system is in a state S which is a continuous variable and emits one value $V = S + E$, where E is an error that follows a probability distribution that does not depend on time. The system transitions to a new state NextS at time $t + 1$, with $\text{NextS} = S + X$ where X is also an error that follows a probability distribution that does not depend on time. Kalman filters have a wide variety of applications, especially in the estimation of trajectories of physical systems. Kalman filters differ from Hidden Markov Model (HMM) because the state and output are continuous instead of discrete.

The program below³, adapted from [Islam et al., 2012b], encodes a Kalman filter in `cplint`:

```

kf(N, O, T) ← init(S), kf_part(0, N, S, O, _LS, T).
kf_part(I, N, S, [V|RO], [S|LS], T) ← I < N, NextI is I + 1,
    trans(S, I, NextS), emit(NextS, I, V),
    kf_part(NextI, N, NextS, RO, LS, T).
kf_part(N, N, S, [], [], S).
trans(S, I, NextS) ←
    {NextS ::= E + S}, trans_err(I, E).
emit(S, I, V) ← {V ::= S + X}, obs_err(I, X).
init(S) : gaussian(S, 0, 1).
trans_err(_, E) : gaussian(E, 0, 2).
obs_err(_, E) : gaussian(E, 0, 1).

```

$\text{kf}(N, O, T)$ means that the filter run for N time points produced the output sequence O and state sequence T starting from state 0.

Continuous random variables appear in arithmetic expressions (in clauses for `trans/3` and `emit/3`). It is often convenient, as in this case, to use CLP(R) constraints as in this way the expressions can be used in multiple directions and the same clauses can be used both to sample and to evaluate the weight of the sample on the basis of evidence (see Section 7.5).

³http://cplint.eu/e/kalman_filter.pl

For example, the expression $\{NextS ::= E + S\}$ can be used to compute the value of any variable given values for the other two.

The semantics is given in terms of a stochastic T_P operator as the one of DCs.

Definition 30 (ST_P operator – cplint). *Let \mathcal{P} be a program and $ground(\mathcal{P})$ be the set of all instances of clauses in \mathcal{P} with all variables in the body replaced by constants. Starting from a set of ground facts I , the ST_P operator returns*

$$\begin{aligned} ST_P(I) = & \{h' \mid h : Density \leftarrow b_1, \dots, b_n \in ground(P) \wedge \forall b_i : b_i \in I \vee \\ & h' = h\{Var/v\} \text{ with } Var \text{ the continuous variable of } h \text{ and} \\ & v \text{ sampled from } Density\} \cup \\ & \{h \mid Dist \leftarrow b_1, \dots, b_n \in ground(P) \wedge \forall b_i : b_i \in I \\ & \text{with } h \text{ sampled from discrete distribution } Dist\} \end{aligned}$$

Differently from ST_P of definition 30, there is no need for special treatment for the atoms in the body, as they are all logical atoms. For each probabilistic clause $h : Density \leftarrow b_1 \dots, b_n$ whenever the body $b_1 \dots, b_n$ is true in I , a value v for the continuous variable Var of h is sampled from the distribution $Density$ and $h\{Var/v\}$ is added to the interpretation. Similarly for discrete and deterministic clauses.

cplint also allows the syntax of DC and Extended PRISM by translating clauses in these languages to cplint clauses.

For DC, this amounts to replacing head atoms of the form $p(t_1, \dots, t_n) \sim density(par_1, \dots, par_m)$ with $p(t_1, \dots, t_n, Var) : density(Var, t_1, \dots, t_n)$ and body atoms of the form $p(t_1, \dots, t_n) \sim = X$ with $p(t_1, \dots, t_n, X)$. On the other hand, terms of the form $\simeq(h)$ for h a random variable are not allowed, and the value of a random variable can be used by unifying it with a logical variable using $h \sim = X$.

For extended PRISM, atoms of the form $msw(p(t_1, \dots, t_n), val)$ in the body of clauses defined by a directive such as

$$\leftarrow set_sw(p(t_1, \dots, t_n), density(par_1, \dots, par_m)).$$

are replaced by probabilistic facts of the form

$$p(t_1, \dots, t_n, Var) : density(Var, par_1, \dots, par_m).$$

The syntax of `cplint` allows a considerable freedom, allowing to express both constraints on random variables and a wide variety of distributions. No syntactic checks are made on the program to ensure validity. In case for example random variables are not sufficiently instantiated to exploit expressions for inferring the values of other variables, inference will return an error.

4.5 Probabilistic Constraint Logic Programming

Michels et al. [2013, 2015] and Michels [2016] proposed the language of PCLP that allows continuous random variables and complex constraints on them. PCLP differs from hybrid ProbLog because it allows constraints that are more general than comparing a continuous random variable with a constant. In this sense, it is more similar to DCs. However, DCs allow *generative definitions*, i.e., definition where a parameter of a distribution could depend on the value of another one, while PCLP doesn't. PCLP provides an alternative definition for hybrid programs that is not based on a stochastic T_P operator but on an extension of Sato's DS. Moreover, PCLP allows the specification of imprecise probability distributions by means of *credal sets*: instead of specifying exactly the probability distribution from which a variable is sampled; a set of probability distributions (a credal set) is specified that includes the unknown correct distribution.

In PCLP, a program \mathcal{P} is split into a set of rules \mathcal{R} and a set of facts \mathcal{F} . The facts define the random variables and the rules define the truth value of the atoms in the Herbrand base of the program given the values of the random variables. The set of random variables is countable $\mathbf{X} = \{X_1, X_2, \dots\}$ and each has a range $Range_i$ that is not limited to be Boolean but can be a general set, for example, \mathbb{N} or \mathbb{R} or even \mathbb{R}^n .

The sample space $W_{\mathbf{X}}$ is given by

$$W_{\mathbf{X}} = Range_1 \times Range_2 \times \dots$$

The event space $\Omega_{\mathbf{X}}$ is user-defined but it should be a σ -algebra. A probability measure $\mu_{\mathbf{X}}$ is also given such that $(W_{\mathbf{X}}, \Omega_{\mathbf{X}}, \mu_{\mathbf{X}})$ is a probability space.

A constraint is a predicate φ that takes values of the random variables as argument, i.e., it is a function from $\{(x_1, x_2, \dots) \mid x_1 \in Range_1, x_2 \in Range_2, \dots\}$ to $\{0, 1\}$. Given a constraint φ , its *constraint solution space* $CSS(\varphi)$ is the set of samples where the constraint holds:

$$CSS(\varphi) = \{\mathbf{x} \in W_{\mathbf{X}} \mid \varphi(\mathbf{x})\}$$

Definition 31 (Probabilistic Constraint Logic Theory [Michels et al., 2015]).
A Probabilistic Constraint Logic Theory \mathcal{P} is a tuple

$$(\mathbf{X}, W_{\mathbf{X}}, \Omega_{\mathbf{X}}, \mu_{\mathbf{X}}, Constr, \mathcal{R})$$

where

- \mathbf{X} is a countable set of random variables $\{X_1, X_2, \dots\}$ each with range $Range_i$;
- $W_{\mathbf{X}} = Range_1 \times Range_2 \times \dots$ is the sample space;
- $\Omega_{\mathbf{X}}$ is the event space, a σ -algebra;
- $\mu_{\mathbf{X}}$ is a probability measure such that $(W_{\mathbf{X}}, \Omega_{\mathbf{X}}, \mu_{\mathbf{X}})$ is a probability space;
- $Constr$ is a set of constraints closed under conjunction, disjunction, and negation such that the constraint solution space of each constraint is included in $\Omega_{\mathbf{X}}$:

$$\{CSS(\varphi) \mid \varphi \in Constr\} \subseteq \Omega_{\mathbf{X}};$$

- \mathcal{R} is a set of logical rules with constraints:

$$h \leftarrow l_1, \dots, l_n, \langle \varphi_1(\mathbf{X}) \rangle, \dots, \langle \varphi_m(\mathbf{X}) \rangle$$

where $\varphi_i \in Constr$ and $\langle \varphi_i(\mathbf{X}) \rangle$ is called constraint atom for $1 \leq i \leq m$.

Random variables are expressed similarly to DC, for example,

$$\text{time_comp}_1 \sim \text{exp}(1)$$

state that time_comp_1 is a random variable with an exponential distribution with rate 1.

Example 60 (Fire on a ship [Michels et al., 2015]). From [Michels et al., 2015]:

Suppose there is a fire in one compartment of a ship. The heat causes the hull of that compartment to warp and if the fire is not extinguished within 1.25 minutes the hull will breach. After 0.75 minutes the fire will spread to the compartment behind. This means that if the fire is extinguished within 0.75 minutes the ship is saved for sure:

$$\text{saved} \leftarrow \langle \text{time_comp}_1 < 0.75 \rangle$$

In the other compartment the hull will breach 0.625 minutes after the fire breaks out. In order to reach the second compartment the fire in the first one has to be extinguished. So both fires have to be extinguished within $0.75 + 0.625 = 1.375$ minutes. Additionally, the fire in the first compartment has to be extinguished within 1.25 minutes, because otherwise the hull breaches there. The second compartment is however more accessible, such that four fire-fighters can extinguish the fire at the same time, which means they can work four times faster:

$$\text{saved} \leftarrow \langle \text{time_comp}_1 < 1.25 \rangle, \\ \langle \text{time_comp}_1 + 0.25 \cdot \text{time_comp}_2 < 1.375 \rangle$$

Finally, assume exponential distributions for both time durations available to extinguish the fires:

$$\text{time_comp}_1 \sim \text{exp}(1) \\ \text{time_comp}_2 \sim \text{exp}(1)$$

The interesting question here is how likely it is that the ship is saved, i.e., $P(\text{saved})$ is required.

This example is a probabilistic constraint logic theory where

$$\mathbf{X} = \{\text{time_comp}_1, \text{time_comp}_2\},$$

Range₁ = Range₂ = \mathbb{R} , the constraint language includes linear inequalities, and the probability measure makes the two variables independent and distributed according to an exponential distribution.

A probability distribution over the logical atoms of the program (the elements of the Herbrand base $\mathcal{B}_{\mathcal{P}}$) is defined as in PRISM: the logical atoms form a countable set of Boolean random variables $\mathbf{Y} = \{Y_1, Y_2, \dots\}$ and the sample space is $W_{\mathbf{Y}} = \{(y_1, y_2, \dots) \mid y_i \in \{0, 1\}, i = 1, 2, \dots\}$. Michels et al. [2015] state that the event space $\Omega_{\mathbf{Y}}$ is the powerset of $W_{\mathbf{Y}}$, so everything is measurable. However, a measure on the powerset is problematic, so a smaller σ -algebra such as that proposed in Sections 3.2 and 3.3 is preferable.

The sample space for the entire theory is $W_{\mathcal{P}} = W_{\mathbf{X}} \times W_{\mathbf{Y}}$ and the event space is the product σ -algebra (see Definition 14), the σ -algebra generated by the products of elements of $\Omega_{\mathbf{X}}$ and $\Omega_{\mathbf{Y}}$:

$$\Omega_{\mathcal{P}} = \Omega_{\mathbf{X}} \otimes \Omega_{\mathbf{Y}} = \sigma(\{\omega_{\mathbf{X}} \times \omega_{\mathbf{Y}} \mid \omega_{\mathbf{X}} \in \Omega_{\mathbf{X}}, \omega_{\mathbf{Y}} \in \Omega_{\mathbf{Y}}\})$$

We now define a probability measure $\mu_{\mathcal{P}}$ that, together with $W_{\mathcal{P}}$ and $\Omega_{\mathcal{P}}$, forms probability space $(W_{\mathcal{P}}, \Omega_{\mathcal{P}}, \mu_{\mathcal{P}})$. Given $w_{\mathbf{X}}$, the set *satisfiable*($w_{\mathbf{X}}$)

contains all the constraints from *Constr* satisfied in sample $w_{\mathbf{X}}$. Thus, $w_{\mathbf{X}}$ determines a logic theory $\mathcal{R} \cup \text{satisfiable}(w_{\mathbf{X}})$ that must have a unique model denoted with $M_{\mathcal{P}}(w_{\mathbf{X}})$. Then probability measure $\mu_{\mathcal{P}}(\omega_{\mathcal{P}})$ for $\omega_{\mathcal{P}} \in \Omega_{\mathcal{P}}$ is defined by considering the event of $\Omega_{\mathbf{X}}$ identified by $\omega_{\mathcal{P}}$:

$$\mu_{\mathcal{P}}(\omega_{\mathcal{P}}) = \mu_{\mathbf{X}}(\{w_{\mathbf{X}} | (w_{\mathbf{X}}, w_{\mathbf{Y}}) \in \omega_{\mathcal{P}}, M_{\mathcal{P}}(w_{\mathbf{X}}) \models w_{\mathbf{Y}}\}) \quad (4.4)$$

Michels et al. [2015] state that $\{w_{\mathbf{X}} | (w_{\mathbf{X}}, w_{\mathbf{Y}}) \in \omega_{\mathcal{P}}, M_{\mathcal{P}}(w_{\mathbf{X}}) \models w_{\mathbf{Y}}\}$ is measurable, i.e., that it belongs to $\Omega_{\mathbf{X}}$; however, the statement is not obvious if $\Omega_{\mathbf{X}}$ is not the powerset of $W_{\mathbf{X}}$ and should be proved as in Section 3.2.

The probability of a query q (a ground atom) is then given by

$$P(q) = \mu_{\mathcal{P}}(\{\omega_{\mathcal{P}} | \omega_{\mathcal{P}} \models q\}) \quad (4.5)$$

If we define the *solution event* $SE(q)$ as

$$SE(q) = \{w_{\mathbf{X}} \in W_{\mathbf{X}} | M_{\mathcal{P}}(w_{\mathbf{X}}) \models q\}$$

then $P(q) = \mu_{\mathbf{X}}(SE(q))$.

Example 61 (Probability of fire on a ship [Michels et al., 2015]). *Continuing Example 60*, $w_{\mathbf{X}} = (\text{time_comp}_1, \text{time_comp}_2) = (x_1, x_2)$, $w_{\mathbf{Y}} = \text{saved} = y_1$ with $\text{Range}_1 = \text{Range}_2 = [0, +\infty)$, and $y_1 \in \{0, 1\}$. So from Equation (4.5) we get

$$P(\text{saved}) = \mu_{\mathcal{P}}(\{(x_1, x_2, y_1) \in W_{\mathcal{P}} | y_1 = 1\})$$

and from Equation (4.4):

$$P(\text{saved}) = \mu_{\mathbf{X}}(\{(x_1, x_2) | (x_1, x_2, y_1) \in W_{\mathcal{P}}, M_{\mathcal{P}}((x_1, x_2)) \models y_1 = 1\})$$

The solution event is

$$SE(\text{saved}) = \{(x_1, x_2) | x_1 < 0.75 \vee (x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375)\}$$

so

$$P(\text{saved}) = \mu_{\mathbf{X}}(\{(x_1, x_2) | x_1 < 0.75 \vee (x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375)\})$$

Since the two constraints $\varphi_1 = x_1 < 1.25$ and $\varphi_2 = x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375$ are not mutually exclusive, we can use the formula $\mu_{\mathbf{X}}(\varphi_1 \vee \varphi_2) = \mu_{\mathbf{X}}(\varphi_1) + \mu_{\mathbf{X}}(\neg\varphi_1 \wedge \varphi_2)$ with

$$\begin{aligned} \neg\varphi_1 \wedge \varphi_2 &= 0.75 < x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375 = \\ &0.75 < x_1 < 1.25 \wedge x_2 < 5.5 - 4x_1 \end{aligned}$$

Knowing that X_1 and X_2 are distributed according to an exponential distribution with parameter 1 (density $p(x) = e^{-x}$), we get

$$\begin{aligned}\mu_{\mathbf{X}}(\varphi_1) &= \int_0^{\infty} p(x_1)I_{\varphi_1}^1(x_1)dx_1 \\ \mu_{\mathbf{X}}(\neg\varphi_1 \wedge \varphi_2) &= \int_0^{\infty} \int_0^{\infty} p(x_1)p(x_2)I_{\neg\varphi_1 \wedge \varphi_2}^1(x_1, x_2)dx_1dx_2\end{aligned}$$

where $I_{\varphi_1}^1(x_1)$ and $I_{\neg\varphi_1 \wedge \varphi_2}^1(x_1, x_2)$ are indicator functions that take value 1 if the respective constraint is satisfied and 0 otherwise. So

$$\begin{aligned}\mu_{\mathbf{X}}(\varphi_1) &= \int_0^{0.75} p(x_1)dx_1 = 1 - e^{-0.75} \approx 0.53 \\ \mu_{\mathbf{X}}(\neg\varphi_1 \wedge \varphi_2) &= \int_{0.75}^{1.25} f(x_1) \left(\int_0^{5.5-4x_1} f(x_2)dx_2 \right) dx_1 = \\ &= \int_{0.75}^{1.25} e^{-x_1} (1 - e^{-5.5+4x_1}) dx_1 = \\ &= \int_{0.75}^{1.25} e^{-x_1} - e^{-5.5+3x_1} dx_1 = \\ &= -e^{-1.25} + e^{-0.75} - \frac{e^{-5.5+3 \cdot 1.25}}{3} + \frac{e^{-5.5+3 \cdot 0.75}}{3} \approx \\ &0.14\end{aligned}$$

So

$$P(\text{saved}) \approx 0.53 + 0.14 \approx 0.67$$

Proposition 5 (Conditions for exact inference [Michels et al., 2015]). *The probability of an arbitrary query can be computed exactly if the following conditions hold:*

1. *Finite-relevant-constraints condition: There are only finitely many constraint atoms that are relevant for each query atom (see Section 1.3) and finding them and checking entailment can be done in finite time.*
2. *Finite-dimensional-constraints condition: Each constraint puts a condition only on a finite number of variables.*
3. *Computable-measure condition: It is possible to compute the probability of finite-dimensional events, i.e., finite-dimensional integrals over the probability density of the random variables.*

The example

$forever_sun(X) \leftarrow \langle Weather_X = sunny \rangle, forever_sun(X + 1)$

does not fulfill the finite-relevant-constraints condition as the set of relevant constraints for $forever_sun(0)$ is $\langle Weather_X = sunny \rangle$ for $X = 0, 1, \dots$

4.5.1 Dealing with Imprecise Probability Distributions

In order to relax the conditions for exact inference of Proposition 5, Michels et al. [2015] consider the problem of computing bounds on the probability of queries.

Credal sets, see Section 2.9, are sets of probability distributions. They can be defined by assigning probability mass to sets of values without specifying exactly how the mass is distributed over those values. For example, for a continuous random variable, we can assign some probability mass to the set of all values between 1 and 3. This mass can be distributed uniformly over the entire set or uniformly over only parts of it or distributed in a more complex manner.

Definition 32 (Credal set specification). A credal set specification \mathbf{C} is a sequence of finite-dimensional credal set specifications C_1, C_2, \dots . Each C_k is a finite set of probability-event pairs $(p_1, \omega_1), (p_2, \omega_2), \dots, (p_n, \omega_n)$ such that for each C_k

1. The events belong to a finite-dimensional event space $\Omega_{\mathbf{X}}^k$ over the sample space $W_{\mathbf{X}} = Range_1 \times Range_2 \times \dots \times Range_k$.
2. The sum of the probabilities is 1.0: $\sum_{(p, \omega) \in C_k} p = 1.0$.
3. The events must not be the empty set $\forall (p, \omega) \in C_k : \omega \neq \emptyset$.

Moreover, C_k must be compatible, i.e., for all k $C_k = \pi_k(C_{k+1})$ where $\pi_l(C_k)$ for $l < k$ is defined as

$$\pi_l(C_k) = \left\{ \left(\sum_{(p, \omega) \in C_k, \pi_l(\omega) = \omega'} p, \omega' \right) \mid \omega' \in \{ \pi_l(\omega) \mid (p, \omega) \in C_k \} \right\}$$

and $\pi_l(\omega)$ is the projection of event ω over the first l components

$$\pi_l(\omega) = \{ (x_1, \dots, x_l) \mid (x_1, \dots, x_l, \dots) \in \omega \}$$

Each C_k identifies a set of probability measures $\Upsilon_{\mathbf{X}}^k$ such that, for each measure $\mu_{\mathbf{X}} \in \Upsilon_{\mathbf{X}}^k$ on $\Omega_{\mathbf{X}}^k$ and each event $\omega \in \Omega_{\mathbf{X}}^k$, the following holds:

$$\sum_{(p, \psi) \in C_k, \psi \subseteq \omega} p \leq \mu_{\mathbf{X}}(\omega) \leq \sum_{(p, \psi) \in C_k, \psi \cap \omega \neq \emptyset} p$$

In fact, the probability mass of events completely included in event ω certainly contribute to the probability of ω , so they are in its lower bound, while the probability of events with a non-empty intersection may fully contribute to the probability, so they are in its upper bound.

Michels et al. [2015] show that, under mild conditions, a credal set specification \mathbf{C} identifies a credal set of probability measures $\Upsilon_{\mathbf{X}}$ over the space $(W_{\mathbf{X}}, \Omega_{\mathbf{X}})$ such that all measures $\mu_{\mathbf{X}}$ of $\Upsilon_{\mathbf{X}}$ agree with each C_i in \mathbf{C} . Moreover, this credal set can be extended to credal set $\Upsilon_{\mathcal{P}}$ of probability measures over the whole program \mathcal{P} , which in turns originates a set \mathbf{P} of probability distributions over queries.

Given a credal set specification for a program \mathcal{P} , we want to compute the lower and upper bounds on the probability of a query q defined as:

$$\begin{aligned} \underline{P}(q) &= \min_{\mu_{\mathcal{P}} \in \Upsilon_{\mathcal{P}}} \mu_{\mathcal{P}}(q) \\ \overline{P}(q) &= \max_{\mu_{\mathcal{P}} \in \Upsilon_{\mathcal{P}}} \mu_{\mathcal{P}}(q) \end{aligned}$$

These bounds can be computed as indicated by the next proposition.

Proposition 6 (Computation of probability bounds). *Given a finite-dimensional credal set specification C_k , the lower and upper probability bounds of a query q fulfilling the finite-dimensional-constraints condition are given by:*

$$\begin{aligned} \underline{P}(q) &= \sum_{(p, \omega) \in C_k, \omega \subseteq SE(q)} p \\ \overline{P}(q) &= \sum_{(p, \omega) \in C_k, \omega \cap SE(q) \neq \emptyset} p \end{aligned}$$

Example 62 (Credal set specification – continuous variables). *Consider example 60 and the following finite-dimensional credal set specification*

$$\begin{aligned} C_2 = \{ & (0.49, \{(x_1, x_2) | 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1\}), \\ & (0.14, \{(x_1, x_2) | 1 \leq x_1 \leq 2, 0 \leq x_2 \leq 1\}), \\ & (0.07, \{(x_1, x_2) | 2 \leq x_1 \leq 3, 0 \leq x_2 \leq 1\}), \\ & (0.14, \{(x_1, x_2) | 0 \leq x_1 \leq 1, 1 \leq x_2 \leq 2\}), \\ & (0.04, \{(x_1, x_2) | 1 \leq x_1 \leq 2, 1 \leq x_2 \leq 2\}), \\ & (0.02, \{(x_1, x_2) | 2 \leq x_1 \leq 3, 1 \leq x_2 \leq 2\}), \\ & (0.07, \{(x_1, x_2) | 0 \leq x_1 \leq 1, 2 \leq x_2 \leq 3\}), \end{aligned}$$

$$(0.02, \{(x_1, x_2) | 1 \leq x_1 \leq 2, 2 \leq x_2 \leq 3\}),$$

$$(0.01, \{(x_1, x_2) | 2 \leq x_1 \leq 3, 2 \leq x_2 \leq 3\})\}$$

Figure 4.1 shows how the probability mass is distributed over the (x_1, x_2) plane. The solution event for $q = \text{saved}$ is

$$SE(\text{saved}) = \{(x_1, x_2) | x_1 < 0.75 \vee (x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375)\}$$

and corresponds to the area of Figure 4.1 to the left of the solid line. We can see that the first event, $0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1$, is such that $CSS(0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1) \subseteq SE(\text{saved})$, so its probability is part of the lower bound.

The next event $1 \leq x_1 \leq 2 \wedge 0 \leq x_2 \leq 1$ instead is not a subset of $SE(\text{saved})$ but has a non-empty intersection with $SE(\text{saved})$, so the probability of the event is part of the upper bound.

Event $2 \leq x_1 \leq 3 \wedge 0 \leq x_2 \leq 1$ instead has an empty intersection with $SE(\text{saved})$, so its probability is not part of any bound.

Of the following events, $0 \leq x_1 \leq 1 \wedge 1 \leq x_2 \leq 2$, $1 \leq x_1 \leq 2 \wedge 1 \leq x_2 \leq 2$, and $0 \leq x_1 \leq 1 \wedge 2 \leq x_2 \leq 3$ have a non-empty intersection with $SE(\text{saved})$ while the remaining have an empty intersection, so overall

$$\underline{P}(q) = 0.49$$

$$\overline{P}(q) = 0.49 + 0.14 + 0.14 + 0.04 + 0.07 = 0.88$$

Credal set specifications can also be used for discrete distributions when the information we have on them is imprecise.

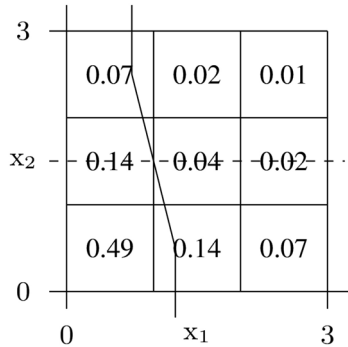


Figure 4.1 Credal set specification for Examples 62 and 64.

Example 63 (Credal set specification – discrete variables). *Suppose you have a model with a single random variable X_1 with $\text{Range}_1 = \{\text{sun}, \text{rain}\}$ representing the weather tomorrow. A credal set specification may consist of*

$$C_1 = \{(0.2, \{\text{sun}, \text{rain}\}), (0.8, \{\text{sun}\})\}$$

The probability distributions that are included in this credal set are those of the form

$$\begin{aligned} P(X_1 = \text{sun}) &= 0.8 + \gamma \\ P(X_1 = \text{rain}) &= 0.2 - \gamma \end{aligned}$$

for $\gamma \in [0, 0.2]$. Thus, the probability that tomorrow is sunny is greater than 0.8 but we don't know its precise value.

In the general case of an infinite dimensional credal set specification, the following holds:

$$\begin{aligned} \underline{P}(q) &= \lim_{k \rightarrow \infty} \sum_{(p, \omega) \in C_k, \omega \subseteq SE(q)} p \\ \overline{P}(q) &= \lim_{k \rightarrow \infty} \sum_{(p, \omega) \in C_k, \omega \cap SE(q) \neq \emptyset} p \end{aligned}$$

A consequence of this is that

$$\begin{aligned} \underline{P}(q) &= 1 - \overline{P}(\sim q) \\ \overline{P}(q) &= 1 - \underline{P}(\sim q) \end{aligned}$$

We can now consider the problem of computing bounds for conditional probabilities. We first define them.

Definition 33 (Conditional probability bounds). *The lower and upper conditional probability bounds of a query q given evidence e are defined as:*

$$\begin{aligned} \underline{P}(q|e) &= \min_{P \in \mathbf{P}} P(q|e) \\ \overline{P}(q|e) &= \max_{P \in \mathbf{P}} P(q|e) \end{aligned}$$

Note that these formulas are different from those of Section 2.9.

Bounds for conditional probabilities can be computed using the following proposition.

Proposition 7 (Conditional probability bounds formulas [Michels et al., 2015]). *The lower and upper conditional probability bounds of a query q are determined by:*

$$\underline{P}(q|e) = \frac{\underline{P}(q, e)}{\underline{P}(q, e) + \overline{P}(\sim q, e)}$$

$$\overline{P}(q|e) = \frac{\overline{P}(q, e)}{\overline{P}(q, e) + \underline{P}(\sim q, e)}$$

Example 64 (Conditional probability bounds). *Consider Example 62 and add the rule $e \leftarrow \langle \text{time_comp}_2 < 1.5 \rangle$.*

Suppose the query is $q = \text{saved}$ and the evidence is e . To compute the lower and upper bounds for $P(q|e)$, we need to compute lower and upper bounds for $q \wedge e$ and $\sim q \wedge e$. The solution event for $SE(e)$ is

$$SE(e) = \{(x_1, x_2) | x_2 < 1.5\}$$

and is shown in Figure 4.1 as the area below the dashed line. The solution event for $q \wedge e$ is

$$SE(q \wedge e) = \{(x_1, x_2) | x_2 < 1.5 \wedge (x_1 < 0.75 \vee (x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375))\}$$

and for $\sim q \wedge e$ is

$$SE(\sim q \wedge e) = \{(x_1, x_2) | x_2 < 1.5 \wedge \neg(x_1 < 0.75 \vee (x_1 < 1.25 \wedge x_1 + 0.25 \cdot x_2 < 1.375))\}$$

We can see that the first event, $0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1$, is such that $CSS(0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1) \subseteq SE(q \wedge e)$, so its probability contributes to $\underline{P}(q \wedge e)$ but not to $\underline{P}(\sim q \wedge e)$.

The next event $1 \leq x_1 \leq 2 \wedge 0 \leq x_2 \leq 1$ has a non-empty intersection with $SE(q)$ and is included in $SE(e)$, so the probability of the event contributes to $\overline{P}(q \wedge e)$ and $\overline{P}(\sim q \wedge e)$.

Event $2 \leq x_1 \leq 3 \wedge 0 \leq x_2 \leq 1$ instead has an empty intersection with $SE(q)$ and is included in $SE(e)$, so it contributes to $\underline{P}(\sim q \wedge e)$ but not to $\underline{P}(q \wedge e)$.

The event $0 \leq x_1 \leq 1 \wedge 1 \leq x_2 \leq 2$ has a non-empty intersection with $SE(q \wedge e)$ and an empty one with $SE(\sim q \wedge e)$, so its probability is part of $\overline{P}(q \wedge e)$.

The event $1 \leq x_1 \leq 2 \wedge 1 \leq x_2 \leq 2$ has a non-empty intersection with $SE(q \wedge e)$ and $SE(\sim q \wedge e)$, so its probability is part of $\underline{P}(q \wedge e)$ and $\underline{P}(\sim q \wedge e)$.

The event $2 \leq x_1 \leq 3 \wedge 1 \leq x_2 \leq 2$ has a non-empty intersection with $SE(\sim q \wedge e)$, so it contributes to $\underline{P}(\sim q \wedge e)$.

The remaining events are included in $SE(\sim e)$, so they are not part of any bound. Overall, we have

$$\begin{aligned} \underline{P}(q \wedge e) &= 0.49 \\ \underline{P}(q \wedge e) &= 0.49 + 0.14 + 0.14 + 0.04 = 0.81 \\ \underline{P}(\sim q \wedge e) &= 0.07 \\ \underline{P}(\sim q \wedge e) &= 0.14 + 0.07 + 0.04 + 0.02 = 0.27 \\ \underline{P}(q|e) &= \frac{0.49}{0.49 + 0.27} \approx 0.64 \\ \underline{P}(q|e) &= \frac{0.81}{0.81 + 0.07} \approx 0.92 \end{aligned}$$

Michels et al. [2015] prove the following theorem that states the conditions under which exact inference of probability bounds is possible.

Theorem 12 (Conditions for exact inference of probability bounds [Michels et al., 2015]). *The probability bounds of an arbitrary query can be computed in finite time under the following conditions:*

1. *Finite-relevant-constraints condition: as condition 1 of Proposition 5.*
2. *Finite-dimensional-constraints condition: as condition 2 of Proposition 5.*
3. *Disjoint-events-decidability condition: for two finite-dimensional events ω_1 and ω_2 in the event space $\Omega_{\mathbf{X}}$, one can decide whether they are disjoint or not ($\omega_1 \cap \omega_2 = \emptyset$).*

Credal sets can be used to approximate continuous distributions arbitrarily well. One has to provide a credal set specification dividing the domain of the variable X_i in n intervals:

$$\{(P(l_1 < X_i < u_1), l_1 < X_i < u_1), \dots, \\ (P(l_n < X_i < u_n), l_n < X_i < u_n)\}$$

with $l_j \leq u_j$, $l_j \in \mathbb{R} \cup \{-\infty\}$ and $u_j \in \mathbb{R} \cup \{+\infty\}$ for $j = 1, \dots, n$. $P(l_j < X_i < u_j)$ must be such that $P(l_j < X_i < u_j) = \int_{l_j}^{u_j} p(x_i) dx_i = F(u_j) - F(l_j)$ where $p(x_i)$ is the probability density of X_i and $F(x_i)$

is its cumulative distribution. The more intervals we provide, the better we approximate the continuous distribution. The probability bounds of the query provide the maximum error of the approximation.

PCLP fixes a syntax for expressing probability distributions precisely and approximately with credal set specifications.

The precise definition of a random variable is the same as for DC. So, for example, the random variable `time_comp1` can be represented with

$$\text{time_comp}_1 \sim \text{exponential}(1).$$

PCLP also allows the definition of multidimensional random variables as

$$(X_1, \dots, X_n)(A_1, \dots, A_m) \sim \text{Density}.$$

where X_1, \dots, X_n are random variables represented by terms, A_1, \dots, A_m are logical variables appearing in X_1, \dots, X_n , and *Density* is a probability density, such as *exponential*, *normal*, etc. There is a different multidimensional random variable

$$(X_1, \dots, X_n)(t_1, \dots, t_m)$$

for each tuple of ground terms t_1, \dots, t_m replacing parameters A_1, \dots, A_m . Variables X_1, \dots, X_n can also be used individually in constraints, the set of them is

$$\{X_i(t_1, \dots, t_m) \mid i = 1, \dots, n, (t_1, \dots, t_m) \text{ is a tuple of terms}\}.$$

Credal set (approximate) specifications are represented as

$$(X_1, \dots, X_n)(A_1, \dots, A_m) \sim \{p_1 : \varphi_1, \dots, p_l : \varphi_l\}$$

where p_i is a probability, φ_i is a satisfiable constraint, and $\sum_{i=1}^l p_i = 1$. The p_i s and φ_i s can include logical variables.

Examples of specifications are

$$\text{temperature}(\text{Day}) \sim \{0.2 : \text{temperature} < 0, 0.8 : \text{temperature} > 0\}.$$

$$\begin{aligned} \text{temperature}(\text{Day}) \sim \{0.2 : \text{temperature} < \text{Day}/1000, \\ 0.8 : \text{temperature} > \text{Day}/1000\}. \end{aligned}$$

where *Day* in the second specification must assume numeric values.

In case the program contains multiple definitions for the same random variable, only the first is considered.

PCLP actually defines a family of languages depending on the constraint language $Constr$: a specific language is indicated with $PCLP(Constr)$. The constraint language must have some properties: infinite dimensional probability measures can be constructed from an infinite number of finite ones with increasing dimensionality and the satisfiability of constraints must be decidable, which is equivalent to decidability of the disjointness of events.

An interesting instance is $PCLP(\mathbf{R}, \mathbf{D})$ where the constraints deal with real numbers (\mathbf{R}) and discrete domains (\mathbf{D}), with the restriction that constraints can include only variables of a single type, either real or discrete.

The constraint theory \mathbf{R} is the same as that of CLP over the reals $CLP(\mathbf{R})$: variables can assume real values and the constraints consist of linear equalities and inequalities. The constraint theory \mathbf{D} is similar to that of CLP over finite domains $CLP(\mathbf{FD})$: variables can assume discrete values and the constraints consist of set membership (\in and \notin) and equality ($=$ and \neq). Differently from $CLP(\mathbf{FD})$, the domains can be countably infinite.

A PCLP program defines a credal set specifications $\mathbf{C} = \{C_1, C_2, \dots\}$ by combining the specification of the individual variables. We first fix the enumeration of random variables and denote the set of the definitions for the first n random variables as D_n . The credal set specification \mathbf{C} of the program is defined as:

$$C_n = \pi_n \left(\left\{ (p, CSS(\varphi)) \mid (p, \varphi) \in \hat{\prod}_{d \in D_n} d \right\} \right)$$

where the product of two random variable definitions $d_1 \hat{\times} d_2$ is defined as:

$$d_1 \hat{\times} d_2 = \{(p_1 \cdot p_2, \varphi_1 \wedge \varphi_2) \mid ((p_1, \varphi_1), (p_2, \varphi_2)) \in d_1 \times d_2\}$$

For example, the following random variable definitions yield the credal set specification of Example 62:

$$\text{time_comp}_1 \sim \{0.7 : 0 \leq \text{time_comp}_1 \leq 1, 0.2 : 1 \leq \text{time_comp}_1 \leq 2, \\ 0.1 : 2 \leq \text{time_comp}_1 \leq 3\}$$

$$\text{time_comp}_2 \sim \{0.7 : 0 \leq \text{time_comp}_2 \leq 1, 0.2 : 1 \leq \text{time_comp}_2 \leq 2, \\ 0.1 : 2 \leq \text{time_comp}_2 \leq 3\}$$

The *solution constraint* of a query q is defined as:

$$SC(q) = \bigvee_{\phi \subseteq Constr, M_{\mathcal{P}}(\phi) \models q} \bigwedge \varphi$$

where $M_{\mathcal{P}}(\phi)$ is the model of the theory $\mathcal{R} \cup \{\langle \varphi \rangle | \varphi \in \phi\}$.

Function

$$check : Constr \rightarrow \{sat, unsat, unknown\}$$

checks satisfiability of constraints and returns

- *sat*: The constraint is certainly satisfiable, i.e., there is a solution.
- *unsat*: The constraint is certainly unsatisfiable, i.e., there is no solution.
- *unknown*: Satisfiability could not be decided, i.e., nothing is said about the constraint.

If the constraint theory is decidable, *unknown* is never returned.

Given function *check*, we can compute the lower and upper probability bounds of a query q taking into account only the first n random variables from a PCLP program fulfilling the exact inference conditions, as

$$\begin{aligned} \underline{P}(q) &= \sum_{(p,\varphi) \in C_n, check(\varphi \wedge \neg SC(q)) = unsat} p \\ \overline{P}(q) &= \sum_{(p,\varphi) \in C_n, check(\varphi \wedge SC(q)) = sat} p \end{aligned}$$

If the constraints are partially decidable, we get the following result

$$\begin{aligned} \underline{P}(q) &\geq \sum_{(p,\varphi) \in C_n, check(\varphi \wedge \neg SC(q)) = unsat} p \\ \overline{P}(q) &\leq \sum_{(p,\varphi) \in C_n, check(\varphi \wedge SC(q)) \neq unsat} p \end{aligned}$$

Inference from a PCLP program can take three forms if *Constr* is decidable:

- exact computation of a point probability, if the random variables are all precisely defined;
- exact computation of lower and upper probabilities, if the information about random variables is imprecise, i.e., it is given as credal sets;
- approximate inference with probability bounds, if information about random variables is precise but credal sets are used to approximate continuous distributions.

If *Constr* is partially decidable, we can perform only approximate inference and obtain bounds in all three cases. In particular, in the second case, we obtain a lower bound on the lower probability and an upper bound on the upper probability.

PCLP, despite the name, is only loosely related to other probabilistic logic formalisms based on CLP such as CLP(BN), see Section 2.10.2, or $\text{clp}(\text{pdf}(y))$ [Angelopoulos, 2003] because it uses constraints to denote events and define credal set specifications, while CLP(BN) and $\text{clp}(\text{pdf}(y))$ consider probability distributions as constraints.

5

Exact Inference

Inference includes a variety of tasks. In the following, let q and e be conjunctions of ground literals, respectively, the query and the evidence.

- The EVID task is to compute an unconditional probability $P(e)$, the probability of evidence. This terminology is especially used when $P(e)$ is computed as part of a solution for the COND task. When there is no evidence, we will speak of $P(q)$, the probability of the query.
- In the COND task, we want to compute the conditional probability distribution of the query given the evidence, i.e., compute $P(q|e)$. A related task is CONDATOMS where we are given a set of ground atoms Q and we want to compute $P(q|e)$ for each $q \in Q$.
- The MPE task, or *most probable explanation*, is to find the most likely truth value of all non-evidence atoms given the evidence, i.e., solving the optimization problem $\arg \max_q P(q|e)$ with q being the unobserved atoms, i.e., $Q = \mathcal{B} \setminus E$, where E is the set of atoms appearing in e and q is an assignment of truth values to the atoms in Q .
- The MAP task, or *maximum a posteriori*, is to find the most likely value of a set of non-evidence atoms given the evidence, i.e., finding $\arg \max_q P(q|e)$ where q is a set of ground atoms. MPE is a special case of MAP where $Q \cup E = \mathcal{B}$.
- The DISTR task involves computing the probability distribution or density of the non-ground arguments of a conjunction of literals q , e.g., computing the probability density of X in goal $mix(X)$ of the Gaussian mixture of Example 57. If the argument is a single one and is numeric (integer or real), then EXP is the task of computing the expected value of the argument (see Section 1.5).

Several approaches have been proposed for inference. Exact inference aims at solving the tasks in an exact way, modulo errors of computer floating point arithmetic. Exact inference can be performed in various ways:

dedicated algorithms for special cases, knowledge compilation, conversion to graphical models, or lifted inference. This chapter discusses exact inference approaches except lifted inference which is presented in Chapter 6.

Exact inference is very expensive since it is $\#P$ -complete in general, because that is the cost of inference in the underlying graphical model [Koller and Friedman, 2009, Theorem 9.2]. Therefore, in some cases, it is necessary to perform approximate inference, i.e., finding an approximation of the answer that is cheaper to compute. The main approach for approximate inference is sampling, but there are others such as iterative deepening or bounding. Approximate inference is discussed in Chapter 7.

In Chapter 3, we saw that the semantics for programs with function symbols is given in terms of explanations, i.e., sets of choices that ensure that the query is true. The probability of a query is given as a function of a covering set of explanations, i.e., a set containing all possible explanations for a query.

This definition suggests an inference approach that consists in finding a covering set of explanations and then computing the probability of the query from it.

To compute the probability of the query, we need to make the explanations pairwise incompatible: once this is done, the probability is the result of a summation.

Early inference algorithms such as [Poole, 1993b] and PRISM [Sato, 1995] required the program to be such that it always had a pairwise incompatible covering set of explanations. In this case, once the set is found, the computation of the probability amounts to performing a sum of products. For programs to allow this kind of approach, they must satisfy the assumptions of independence of subgoals and exclusiveness of clauses, which mean that [Sato et al., 2017]:

1. the probability of a conjunction (A, B) is computed as the product of the probabilities of A and B (*independent-and assumption*),
2. the probability of a disjunction $(A; B)$ is computed as the sum of the probabilities of A and B (*exclusive-or assumption*).

See also Section 5.9.

5.1 PRISM

PRISM [Sato, 1995; Sato and Kameya, 2001, 2008] performs inference on programs respecting the assumptions of independent-and and exclusive-or

by means of an algorithm for computing and encoding explanations in a factorized way instead of explicitly generating all explanations. In fact, the number of explanations may be exponential, even if they can be encoded compactly.

Example 65 (Hidden Markov model – PRISM [Sato and Kameya, 2008]).

An Hidden Markov Model (HMM) [Rabiner, 1989] is a dynamical system that, at each integer time point t , is in a state S from a finite set and emits one symbol O according to a probability distribution $P(O|S)$ that is independent of time. Moreover, it transitions to a new state $NextS$ at time $t + 1$, with $NextS$ chosen according to $P(NextS|S)$, again independently of time. HMMs are so called because they respect the Markov condition: the state at time t depends only from the state at time $t - 1$ and is independent of previous states. Moreover, the states are usually hidden: the task is to obtain information on them from the sequence of output symbols, modeling systems that can be only observed from the outside. HMMs and Kalman filters (see Example 59) are similar; they differ because the first uses discrete states and output symbols and the latter continuous ones. HMMs have applications in many fields, such as speech recognition.

The following program encodes an HMM with two states, $\{s1, s2\}$, of which $s1$ is the start state, and two output symbols, a and b :

```

values(tr(s1), [s1, s2]).
values(tr(s2), [s1, s2]).
values(out(-), [a, b]).
hmm(Os) ← hmm(s1, Os).
hmm(_S, []).
hmm(S, [O|Os]) ←

```

```

    msw(out(S), O), msw(tr(S), NextS), hmm(Next, Os).

```

The query $P(hmm(Os))$ asked against this program is the probability that the sequence of symbols Os is emitted.

Note that msw atoms have two arguments here, so each call to such atoms is intended to refer to a different random variable. This means that if the same msw is encountered again in a derivation, it is associated with a different random variable, differently from the other languages under the DS where a ground instance of a probabilistic clause is associated with only one random variable. The latter approach is also called memoing, meaning that the associations between atoms and random variables are stored for reuse, while the approach of PRISM is often adopted by non-logic probabilistic programming languages.

$$\begin{aligned}
E_1 &= m(\text{out}(s1), a), m(\text{tr}(s1), s1), m(\text{out}(s1), b), m(\text{tr}(s1), s1), \\
&\quad m(\text{out}(s1), b), m(\text{tr}(s1), s1), \\
E_2 &= m(\text{out}(s1), a), m(\text{tr}(s1), s1), m(\text{out}(s1), b), m(\text{tr}(s1), s1), \\
&\quad m(\text{out}(s1), b), m(\text{tr}(s1), s2), \\
E_3 &= m(\text{out}(s1), a), m(\text{tr}(s1), s1), m(\text{out}(s2), b), m(\text{tr}(s1), s2), \\
&\quad m(\text{out}(s2), b), m(\text{tr}(s2), s1), \\
&\dots \\
E_8 &= m(\text{out}(s1), a), m(\text{tr}(s1), s2), m(\text{out}(s2), b), m(\text{tr}(s2), s2), \\
&\quad m(\text{out}(s2), b), m(\text{tr}(s2), s2)
\end{aligned}$$

Figure 5.1 Explanations for query $\text{hmm}([a, b, b])$ of Example 65.

Consider the query $\text{hmm}([a, b, b])$ and the problem of computing the probability of output sequence $[a, b, b]$. Such a query has the eight explanations shown in Figure 5.1 where *msw* is abbreviated by *m*, repeated atoms correspond to different random variables, and each explanation is a conjunction of *msw* atoms. In general, the number of explanations is exponential in the length of the sequence.

If the query q has the explanations $E_1 \dots, E_n$, we can build the formula

$$q \Leftrightarrow E_1 \vee \dots \vee E_n$$

expressing the truth of q as a function of the *msw* atoms in the explanations. The probability of q is then given by $P(q) = \sum_{i=1}^n P(E_i)$ and the probability of each explanation is the product of the probability of each atom, because explanations are mutually exclusive, as each explanation differs from the others in the choice for at least one *msw* atom.

PRISM performs inference by deriving the query with tabling and storing, for each subgoal g , the switches and atoms on which g directly depends. In practice, for each subgoal g , PRISM builds a formula

$$g \Leftrightarrow S_1 \vee \dots \vee S_n$$

where each S_i is a conjunction of *msw* atoms and subgoals. For the query $\text{hmm}([a, b, b])$ from the program of Example 65, PRISM builds the formulas shown in Figure 5.2.

Differently from explanations, the number of such formulas is linear rather than exponential in the length of the output sequence.

PRISM assumes that the subgoals in the derivation of q can be ordered $\{g_1, \dots, g_m\}$ such that

$$g_i \Leftrightarrow S_{i1} \vee \dots \vee S_{in_i}$$

$$\begin{aligned}
&hmm([a, b, b]) \Leftrightarrow hmm(s1, [a, b, b]) \\
&hmm(s1, [a, b, b]) \Leftrightarrow m(out(s1), a), m(tr(s1), s1), hmm(s1, [b, b]) \vee \\
&\quad m(out(s1), a), m(tr(s1), s2), hmm(s2, [b, b])) \\
&hmm(s1, [b, b]) \Leftrightarrow m(out(s1), b), m(tr(s1), s1), hmm(s1, [b]) \vee \\
&\quad m(out(s1), b), m(tr(s1), s2), hmm(s2, [b])) \\
&hmm(s2, [b, b]) \Leftrightarrow m(out(s2), b), m(tr(s2), s1), hmm(s1, [b]) \vee \\
&\quad m(out(s2), b), m(tr(s2), s2), hmm(s2, [b])) \\
&hmm(s1, [b]) \Leftrightarrow m(out(s1), b), m(tr(s1), s1), hmm(s1, []) \vee \\
&\quad m(out(s1), b), m(tr(s1), s2), hmm(s2, [])) \\
&hmm(s2, [b]) \Leftrightarrow m(out(s2), b), m(tr(s2), s1), hmm(s1, []) \vee \\
&\quad m(out(s2), b), m(tr(s2), s2), hmm(s2, [])) \\
&hmm(s1, []) \Leftrightarrow true \\
&hmm(s2, []) \Leftrightarrow true
\end{aligned}$$

Figure 5.2 PRISM formulas for query $hmm([a, b, b])$ of Example 65.

where $q = g_1$ and each S_{ij} contains only *msw* atoms and subgoals from $\{g_{i+1}, \dots, g_m\}$. This is called the *acyclic support condition* and is true if tabling succeeds in evaluating q , i.e., if it doesn't go into a loop.

From these formulas, the probability of each subgoal can be obtained by means of Algorithm 3 that computes the probability of each subgoal bottom up and reuses the computed values for subgoals higher up. This is a *dynamic programming* algorithm: the problem is solved by breaking it down into simpler sub-problems in a recursive manner.

Algorithm 3 Function PRISM-PROB: Computation of the probability of a query.

```

1: function PRISM-PROB( $q$ )
2:   for all  $i, k$  do
3:      $P(msw(i, v_k)) \leftarrow \Pi_{ik}$ 
4:   end for
5:   for  $i \leftarrow m \rightarrow 1$  do
6:      $P(g_i) \leftarrow 0$   $\triangleright P(g_i)$  is the probability of goal  $g_i$ 
7:     for  $j \leftarrow 1 \rightarrow n_i$  do
8:       Let  $S_{ij}$  be  $h_1, \dots, h_{ij\circ}$ 
9:        $R(S_{ij}) \leftarrow \prod_{l=1}^{\circ} P(h_l)$   $\triangleright R(S_{ij})$  is the probability of explanation
10:       $S_{ij}$  of goal  $g_i$ 
11:       $P(g_i) \leftarrow P(g_i) + R(S_{ij})$ 
12:    end for
13:  end for
14:  return  $P(q)$ 
15: end function

```

$$\begin{aligned}
P(hmm(s1, [])) &= 1 \\
P(hmm(s2, [])) &= 1 \\
P(hmm(s1, [b])) &= \\
&\quad P(m(out(s1), b)) \cdot P(m(tr(s1), s1)) \cdot P(hmm(s1, [])) + \\
&\quad P(m(out(s1), b)) \cdot P(m(tr(s1), s2)) \cdot P(hmm(s2, [])) \\
P(hmm(s2, [b])) &= \\
&\quad P(m(out(s2), b)) \cdot P(m(tr(s2), s1)) \cdot P(hmm(s1, [])) + \\
&\quad P(m(out(s2), b)) \cdot P(m(tr(s2), s2)) \cdot P(hmm(s2, [])) \\
&\dots
\end{aligned}$$

Figure 5.3 PRISM computations for query $hmm([a, b, b])$ of Example 65.

For the example above, the probabilities are computed as in Figure 5.3.

The cost of computing the probability of the query in this case is thus linear in the length of the output sequence, rather than exponential.

In the case of HMMs, computing the probability of an output sequence using PRISM has the same complexity $O(T)$ as the specialized forward algorithm [Rabiner, 1989], where T is the length of the sequence.

The MPE task can be performed by replacing summation in Algorithm 3 with max and arg max. In the case of HMM, this yields the most likely sequence of states that originated the output sequence, also called the Viterbi path. Such a path is computed for HMMs by the Viterbi algorithm [Rabiner, 1989] and PRISM has the same complexity $O(T)$.

Writing programs satisfying the assumptions of independence of subgoals and exclusiveness of clauses is not easy and significantly limits the modeling power of the language. Therefore, work was dedicated to lifting these limitations, leading to the AILog2 system Poole [2000] that used the splitting algorithm (Algorithm 1), and to the adoption of knowledge compilation.

5.2 Knowledge Compilation

Knowledge compilation [Darwiche and Marquis, 2002] is an approach for solving certain hard inference tasks on a Boolean formula by compiling it into a form on which the tasks are tractable. Clearly, the complexity is not eliminated but moved to the compilation process. The inference tasks of COND, MPE, and EVID can be solved using well-known techniques from the SAT community applied to weighted Boolean formulas, such as techniques for Weighted Model Counting (WMC) or weighted MAX-SAT. For example, EVID reduces to WMC, i.e., computing the sum of the weights of the worlds where the formula is true. WMC is a generalization of the

problem of counting the number of worlds where the query is true, also known as *model counting*. Model counting and WMC are #P-complete in general but can be computed in polynomial time for Boolean formulas in certain forms.

The knowledge compilation approach to inference follows the two-step method where first the program, the query, and the evidence are converted into a Boolean formula encoding a covering set of explanations and then knowledge compilation is applied to the formula.

The formula is a function of Boolean variables, each encoding a choice, that takes value 1 exactly for assignments corresponding to worlds where the query is true. Therefore, to compute the probability of the query, we can compute the probability that the formula takes value 1 knowing the probability distribution of all the variables and that they are pairwise independent. This is done in the second step, which amounts to converting the formula into a form from which computing the probability is easy. The conversion step is called *knowledge compilation* [Darwiche and Marquis, 2002] because it compiles the formula into a special form with the property that the cost of solving the problem is polynomial. This second step is a well-known problem also called *disjoint-sum* and is an instance of WMC.

5.3 ProbLog1

The ProbLog1 system [De Raedt et al., 2007] compiles explanations for queries from ProbLog programs to Binary Decision Diagrams (BDDs). It uses a source to source transformation of the program [Kimmig et al., 2008] that replaces probabilistic facts with clauses that store information about the fact in the dynamic database of the Prolog interpreter. When a successful derivation for the query is found, the set of facts stored in the dynamic database is collected to form a new explanation that is stored away and backtracking is triggered to find other possible explanations.

If K is the set of explanations found for query q , the probability of q is given by the probability of the formula

$$f_K(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(F_i, \theta_j, 1) \in \kappa} X_{ij} \bigwedge_{(F_i, \theta_j, 0) \in \kappa} \neg X_{ij}$$

where X_{ij} is a Boolean random variable associated with grounding $F_i \theta_j$ of fact F_i and $P(X_{ij} = 1) = \Pi_i$.

Consider Example 13 that we repeat here for ease of reading

$$\begin{aligned} sneezing(X) &\leftarrow flu(X), flu_sneezing(X). \\ sneezing(X) &\leftarrow hay_fever(X), hay_fever_sneezing(X). \\ flu(bob). \\ hay_fever(bob). \end{aligned}$$

$$\begin{aligned} F_1 &= 0.7 :: flu_sneezing(X). \\ F_2 &= 0.8 :: hay_fever_sneezing(X), \end{aligned}$$

A set of covering explanations for $sneezing(bob)$ is $K = \{\kappa_1, \kappa_2\}$ with

$$\kappa_1 = \{(F_1, \{X/bob\}, 1)\} \quad \kappa_2 = \{(F_2, \{X/bob\}, 1)\}$$

If we associate X_{11} with $F_1\{X/bob\}$ and X_{21} with $F_2\{X/bob\}$, the Boolean formula is

$$f_K(\mathbf{X}) = X_{11} \vee X_{21} \tag{5.1}$$

In this simple case, the probability that $f_K(\mathbf{X})$ takes value 1 can be computed by using the formula for the probability of a disjunction:

$$P(X_{11} \vee X_{21}) = P(X_{11}) + P(X_{21}) - P(X_{11} \wedge X_{21})$$

and, since X_{11} and X_{21} are independent, we get

$$P(f_K(\mathbf{X})) = P(X_{11} \vee X_{21}) = P(X_{11}) + P(X_{21}) - P(X_{11})P(X_{21}).$$

In the general case, however, simple formulas like this can't be applied.

BDDs are a target language for knowledge compilation. A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node n has two children: one corresponding to the 1 value of the variable associated with the level of n and one corresponding to the 0 value of the variable. When drawing BDDs, the 0-branch is distinguished from the 1-branch by drawing it with a dashed line. The leaves store either 0 or 1. For example, a BDD representing Function 5.1 is shown in Figure 5.4.

Given values for all the variables, the computation of the value of the function can be performed by traversing the BDD starting from the root and returning the value associated with the leaf that is reached.

To compile a Boolean formula $f(\mathbf{X})$ into a BDD, software packages incrementally combine sub-diagram using Boolean operations.

BDDs can be built with various software packages that perform knowledge compilation by providing Boolean operations between diagrams. So the

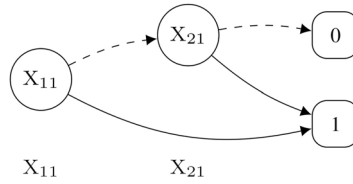


Figure 5.4 BDD representing Function 5.1.

diagram for a formula is obtained by applying the Boolean operations in the formula bottom-up, combining diagrams representing X_{ij} or $\neg X_{ij}$ into progressively more complex diagrams.

After the application of an operation, isomorphic portions of the resulting diagram are merged and redundant nodes are deleted, possibly changing the order of variables if useful. This often allows the diagram to have a number of nodes much smaller than exponential in the number of variables that a naive representation of the function would require.

In the BDD of Figure 5.4, a node for variable X_{21} is absent from the path from the root to the 1 leaf. The node has been deleted because both arcs from the node would go to the 1 leaf, so the node is redundant.

Example 66 (Epidemic – ProbLog). *The following ProbLog program \mathcal{P} encodes a very simple model of the development of an epidemic:*

epidemic \leftarrow *flu*(X), *ep*(X), *cold*.

flu(*david*).

flu(*robert*).

$F_1 = 0.7 :: \textit{cold}$.

$F_2 = 0.6 :: \textit{ep}(X)$.

*This program models the fact that, if somebody has the flu and the climate is cold, there is the possibility that an epidemic arises. $\textit{ep}(X)$ is true if X is an active cause of epidemic and is defined by a probabilistic fact. We are uncertain about whether the climate is cold but we know for sure that David and Robert have the flu. Fact F_1 has one grounding, associated with variable X_{11} , while F_2 has two groundings, associated with variables X_{21} and X_{22} . The query *epidemic* is true if the Boolean formula*

$$f(\mathbf{X}) = X_{11} \wedge (X_{21} \vee X_{22})$$

is true. The BDD representing this formula is shown in Figure 5.5. As you can see, the subtree rooted at X_{22} can be reached by more than one path. In this case, the BDD compilation system recognized the presence of two

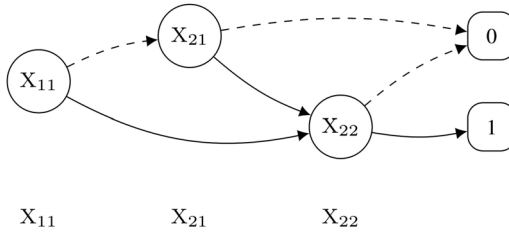


Figure 5.5 BDD for query *epidemic* of Example 66.

isomorphic subgraphs and merged them, besides deleting nodes for X_{21} from the path from X_{11} to X_{22} and for X_{22} from the path from X_{21} to the 0 leaf.

BDDs perform a Shannon expansion of the Boolean formula: they express the formula as

$$f_K(\mathbf{X}) = X_1 \vee f_K^{X_1}(\mathbf{X}) \wedge \neg X_1 \vee f_K^{\neg X_1}(\mathbf{X})$$

where X_1 is the variable associated with the root node of the diagram, $f_K^{X_1}(\mathbf{X})$ is the Boolean formula where X_1 is set to 1, and $f_K^{\neg X_1}(\mathbf{X})$ is the Boolean formula where X_1 is set to 0. The expansion can be applied recursively to the functions $f_K^{X_1}(\mathbf{X})$ and $f_K^{\neg X_1}(\mathbf{X})$.

The formula is thus expressed as a disjunction of mutually exclusive terms, as one contains X_1 and the other $\neg X_1$, so the probability of the formula can be computed with a sum

$$P(f_K(\mathbf{X})) = P(X_1)P(f_K^{X_1}(\mathbf{X})) + (1 - P(X_1))P(f_K^{\neg X_1}(\mathbf{X})).$$

For the BDD of Figure 5.5, this becomes

$$P(f_K(\mathbf{X})) = 0.7 \cdot P(f_K^{X_{11}}(\mathbf{X})) + 0.3 \cdot P(f_K^{\neg X_{11}}(\mathbf{X}))$$

This means that the probability of the formula, and so of the query, can be computed with Algorithm 4: the BDD is recursively traversed and the probability of a node is computed as a function of the probabilities of its children. Note that a table is updated to store the probability of nodes already visited: in fact, BDDs can have multiple paths to a node in case two sub-diagrams are merged and, if a node already visited is encountered again, we can simply retrieve its probability from the table. This ensures that each node is visited exactly once, so the cost of the algorithm is linear in the number of nodes. This is an instance of a dynamic programming algorithm.

Algorithm 4 Function PROB: Computation of the probability of a BDD.

```

1: function PROB(node)
2:   if node is a terminal then
3:     return 1
4:   else
5:     if Table(node)  $\neq$  null then
6:       return Table(node)
7:     else
8:        $p_0 \leftarrow$  PROB(child0(node))
9:        $p_1 \leftarrow$  PROB(child1(node))
10:      let  $\pi$  be the probability of being true of var(node)
11:       $Res \leftarrow p_1 \cdot \pi + p_0 \cdot (1 - \pi)$ 
12:      add node  $\rightarrow$  Res to Table
13:      return Res
14:    end if
15:  end if
16: end function

```

5.4 *cplint*

The *cplint* system (CPLogic INTerpreter) [Riguzzi, 2007a] applies knowledge compilation to LPADs. Differently from ProbLog1, the random variables associated with clauses can have more than two values. Moreover, with *cplint*, programs may contain negation.

To handle multivalued random variables, we can use Multivalued Decision Diagrams (MDDs) [Thayse et al., 1978], an extension of BDDs. Similarly to BDDs, an MDD represents a function $f(\mathbf{X})$ taking Boolean values on a set of multivalued variables \mathbf{X} by means of a rooted graph that has one level for each variable. Each node has one child for each possible value of the multivalued variable associated with the level of the node. The leaves store either 0 or 1. Given values for all the variables \mathbf{X} , we can compute the value of $f(\mathbf{X})$ by traversing the graph starting from the root and returning the value associated with the leaf that is reached.

In order to represent sets of explanations with MDDs, each ground clause $C_i\theta_j$ appearing in the set of explanations is associated with a multivalued variable X_{ij} with as many values as atoms in the head of C_i . Each atomic choice (C_i, θ_j, k) is represented by the propositional equation $X_{ij} = k$. Equations for a single explanation are conjoined and the conjunctions for the different explanations are disjointed. The resulting function takes value 1 if the values taken by the multivalued variables correspond to an explanation for the goal.

Example 67 (Detailed medical symptoms – MDD). *Consider Example 19 that we repeat here for readability:*

$$\begin{aligned}
 C_1 &= \text{strong_sneezing}(X) : 0.3 ; \text{moderate_sneezing}(X) : \\
 &\quad 0.5 \leftarrow \text{flu}(X). \\
 C_2 &= \text{strong_sneezing}(X) : 0.2 ; \text{moderate_sneezing}(X) : \\
 &\quad 0.6 \leftarrow \text{hay_fever}(X). \\
 &\quad \text{flu}(\text{bob}). \\
 &\quad \text{hay_fever}(\text{bob}).
 \end{aligned}$$

A set of explanations for $\text{strong_sneezing}(\text{bob})$ is $K = \{\kappa_1, \kappa_2\}$ with

$$\kappa_1 = \{(C_1, \{X/\text{bob}\}, 1)\} \quad \kappa_2 = \{(C_2, \{X/\text{bob}\}, 1)\}$$

This set of explanations can be represented by the function

$$f_K(\mathbf{X}) = (X_{11} = 1) \vee (X_{21} = 1) \tag{5.2}$$

The corresponding MDD is shown in Figure 5.6.

The probability of the goal is given by the probability of $f_K(\mathbf{X})$ taking value 1.

As BDDs, MDDs represent a Boolean function $f(\mathbf{X})$ by means of a generalization of the Shannon expansion:

$$f(\mathbf{X}) = (X_1 = 1) \wedge f^{X_1=1}(\mathbf{X}) \vee \dots \vee (X_1 = n) \wedge f^{X_1=n}(\mathbf{X})$$

where X_1 is the variable associated with the root node of the diagram and $f^{X_1=k}(\mathbf{X})$ is the function associated with the k -th child of the root node. The expansion can be applied recursively to the functions $f^{X_1=k}(\mathbf{X})$. This expansion allows the probability of $f(\mathbf{X})$ to be expressed by means of the following recursive formula

$$\begin{aligned}
 P(f(\mathbf{X})) &= P(X_1 = 1) \cdot P(f^{X_1=1}(\mathbf{X})) + \dots + \\
 &\quad P(X_1 = n) \cdot P(f^{X_1=n}(\mathbf{X}))
 \end{aligned}$$

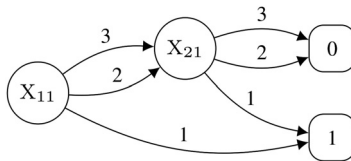


Figure 5.6 MDD for the diagnosis program of Example 19.

because the disjuncts are mutually exclusive due to the presence of the $X_1 = k$ equations. Thus, the probability of $f(\mathbf{X})$ can be computed by means of a dynamic programming algorithm similar to Algorithm 4 that traverses the MDD and sums up probabilities.

Knowledge compilation libraries for MDDs combine diagrams representing equations of the form $X_i = k$ into progressively more complex diagrams using Boolean operations. However, most libraries are restricted to work on BDD, i.e., decision diagrams where all the variables are Boolean. To work on MDDs with a BDD package, we must represent multivalued variables by means of binary variables. Various options are possible, in `cplint` [Riguzzi, 2007a], the choice was to use a binary encoding. A multivalued variable X with n values is encoded with $b = \lceil \log_2 n \rceil$ Boolean variables X_1, \dots, X_b : if $k = k_b \dots k_1$ is the binary encoding of value k , then $X = k$ is encoded as $X_1 = k_1 \wedge \dots \wedge X_b = k_b$.

The dynamic programming algorithm for computing the probability was adjusted to ensure that Boolean variables encoding the same multivalued variable are kept together in the BDD and to correctly retrieve the probability values Π_k by detecting the value k from the configuration of Boolean variables.

`cplint` finds explanations by using a meta-interpreter (see Section 1.3) that resolves the query and keeps a list of the choices encountered during the derivation. Negative goals of the form $\sim a$ are handled by finding a covering set K of explanations for a and then computing their complement, i.e., a set of explanations \overline{K} that identifies all the worlds where a is false. This is done with an algorithm similar to function `DUALS(K)` of Algorithm 2 [Poole, 2000]: an explanation κ in \overline{K} is generated by picking an atomic choice (C_i, θ_j, k) from each explanation of K and inserting in \overline{K} an explanation containing (C_i, θ_j, k') with $k' \neq k$. By doing this in all possible ways, the complement \overline{K} of K is obtained.

The system `PICL` [Riguzzi, 2009] applies this approach to `ICL` and, together with the `cplint` system, formed the initial core of the `cplint` suite of algorithms.

5.5 SLGAD

SLGAD [Riguzzi, 2008a, 2010] was an early system for performing inference from LPADs using a modification of SLG resolution [Chen and Warren, 1996], see Section 1.4.2. SLG resolution is the main inference procedure for

normal programs under the WFS and uses tabling to ensure termination and correctness for a large class of programs.

SLGAD is of interest because it performs inference without using knowledge compilation, exploiting the feature of SLG resolution that answers are added to the table, only once, the first time they are derived, while the following calls to the same subgoal retrieve the answers from the table. So the decision of considering an atom as true is an atomic operation. Since this happens when the body of the grounding of a clause with that atom in the head has been proved true, this corresponds to making a choice regarding the clause grounding. SLGAD modifies SLG by performing backtracking on such choices: when an answer can be added to the table, SLGAD checks whether this is consistent with previous choices and, if so, it adds the current choice to the set of choices of the derivation, adds the answer to the table and leaves a choice point, so that the other choices are explored in backtracking. SLGAD thus returns, for a ground goal, a set of explanations that are mutually exclusive because each backtracking choice is made among incompatible alternatives. Therefore, the probability of the goal can be computed by summing the probability of the explanations.

SLGAD was implemented by modifying the Prolog meta-interpreter of [Chen et al., 1995]. Its comparison with systems performing knowledge compilation, however, showed that SLGAD was slower, probably because it is not able to factor explanations as, for example, PRISM does.

5.6 PITA

The PITA system [Riguzzi and Swift, 2010, 2011, 2013] performs inference for LPADs using knowledge compilation to BDDs.

PITA applies a program transformation to an LPAD to create a normal program that contains calls for manipulating BDDs. In the implementation, these calls provide a Prolog interface to the CUDD¹ [Somenzi, 2015] C library and use the following predicates²

- *init* and *end*: for allocation and deallocation of a BDD manager, a data structure used to keep track of the memory for storing BDD nodes;
- *zero*($-BDD$), *one*($-BDD$), *and*($+BDD1, +BDD2, -BDDO$), *or*($+BDD1, +BDD2, -BDDO$) and *not*($+BDDI, -BDDO$): Boolean operations between BDDs;

¹<http://vlsi.colorado.edu/~fabio/>

²BDDs are represented in CUDD as pointers to their root node.

- $add_var(+N_Val, +Probs, -Var)$: addition of a new multivalued variable with N_Val values and parameters $Probs$;
- $equality(+Var, +Value, -BDD)$: returns BDD representing $Var = Value$, i.e., that the random variable Var is assigned $Value$ in the BDD;
- $ret_prob(+BDD, -P)$: returns the probability of the formula encoded by BDD .

$add_var(+N_Val, +Probs, -Var)$ adds a new random variable associated with a new instantiation of a rule with N_Val head atoms and parameters list $Probs$. The auxiliary predicate $get_var_n/4$ is used to wrap $add_var/3$ and avoids adding a new variable when one already exists for an instantiation. As shown below, a new fact $var(R, S, Var)$ is asserted each time a new random variable is created, where R is an identifier for the LPAD clause, S is a list of constants, one for each variables of the clause, and Var is a integer that identifies the random variable associated with a specific grounding of clause R . The auxiliary predicate has the following definition

```

get_var_n(R, S, Probs, Var) ←
    (var(R, S, Var) →
        true
    ;
        length(Probs, L),
        add_var(L, Probs, Var),
        assert(var(R, S, Var))
    ).

```

where $Probs$ is a list of floats that stores the parameters in the head of rule R . R , S , and $Probs$ are input arguments while Var is an output argument. $assert/1$ is a builtin Prolog predicate that adds its argument to the program, allowing its dynamic extension.

The PITA transformation applies to atoms, literals, conjunction of literals, and clauses. The transformation for an atom a and a variable D , $PITA(a, D)$, is a with the variable D added as the last argument. The transformation for a negative literal $b = \sim a$, $PITA(b, D)$, is the expression

$$(PITA(a, DN) \rightarrow not(DN, D); one(D))$$

which is an if-then-else construct in Prolog: if $PITA(a, DN)$ evaluates to true, then $not(DN, D)$ is called, otherwise $one(D)$ is called.

A conjunction of literals b_1, \dots, b_m becomes:

$$\begin{aligned}
 PITA(b_1, \dots, b_m, D) = & one(DD_0), \\
 & PITA(b_1, D_1), and(DD_0, D_1, DD_1), \dots, \\
 & PITA(b_m, D_m), and(DD_{m-1}, D_m, D).
 \end{aligned}$$

The disjunctive clause $C_r = h_1 : \Pi_1 \vee \dots \vee h_n : \Pi_n \leftarrow b_1, \dots, b_m$, where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r) = \{PITA(C_r, 1), \dots, PITA(C_r, n)\}$ with:

$$PITA(C_r, i) = PITA(h_i, D) \leftarrow PITA(b_1, \dots, b_m, DD_m), \\ get_var_n(r, S, [\Pi_1, \dots, \Pi_n], Var), equality(Var, i, DD), \\ and(DD_m, DD, D).$$

for $i = 1, \dots, n$, where S is a list containing all the variables appearing in C_r .

A non-disjunctive fact $C_r = h$ is transformed into the clause

$$PITA(C_r) = PITA_h(h, D) \leftarrow one(D).$$

A disjunctive fact $C_r = h_1 : \Pi_1 \vee \dots \vee h_n : \Pi_n$, where the parameters sum to 1, is transformed into the set of clauses

$$PITA(C_r) = \{PITA(C_r, 1), \dots, PITA(C_r, n)\}$$

with:

$$PITA(C_r, i) = get_var_n(r, S, [\Pi_1, \dots, \Pi_n], Var), \\ equality(Var, i, DD), and(DD_m, DD, D).$$

for $i = 1, \dots, n$.

In the case where the parameters do not sum to one, the clause is first transformed into $null : 1 - \sum_1^n \Pi_i \vee h_1 : \Pi_1 \vee \dots \vee h_n : \Pi_n$. and then into the clauses above, where the list of parameters is $[1 - \sum_1^n \Pi_i, \Pi_1, \dots, \Pi_n,]$ but the 0-th clause (the one for $null$) is not generated.

The definite clause $C_r = h \leftarrow b_1, b_2, \dots, b_m$. is transformed into the clause

$$PITA(C_r) = PITA(h, D) \leftarrow PITA(b_1, \dots, b_m, D).$$

Example 68 (Medical example – PITA). *Clause C_1 from the LPAD of Example 67 is translated to*

$$strong_sneezing(X, BDD) \leftarrow one(BB_0), flu(X, B_1), \\ and(BB_0, B_1, BB_1), \\ get_var_n(1, [X], [0.3, 0.5, 0.2], Var), \\ equality(Var, 1, B), and(BB_1, B, BDD).$$

$$moderate_sneezing(X, BDD) \leftarrow one(BB_0), flu(X, B_1), \\ and(BB_0, B_1, BB_1), \\ get_var_n(1, [X], [0.3, 0.5, 0.2], Var), \\ equality(Var, 2, B), and(BB_1, B, BDD).$$

while clause C_3 is translated to

$$flu(david, BDD) \leftarrow one(BDD).$$

In order to answer queries, the goal $prob(Goal, P)$ is used, which is defined by

```

prob(Goal, P) ← init, retractall(var(-, -, -)),
  add_bdd_arg(Goal, BDD, GoalBDD),
  (call(GoalBDD) → ret_prob(BDD, P); P = 0.0),
end.

```

Since variables may be multivalued, an encoding with Boolean variables must be chosen. The encoding used by PITA is the same as that used to translate LPADs into ProbLog seen in Section 2.4 that was proposed in [De Raedt et al., 2008].

Consider a variable X_{ij} associated with grounding θ_j of clause C_i having n values. We encode it using $n - 1$ Boolean variables

$$X_{ij1}, \dots, X_{ijn-1}.$$

We represent the equation $X_{ij} = k$ for $k = 1, \dots, n - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \dots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n$ by means of the conjunction $\overline{X_{ij1}} \wedge \dots \wedge \overline{X_{ijn-1}}$. The BDD representation of the function in Equation (5.2) is given in Figure 5.7. The Boolean variables are associated with the following parameters:

$$\begin{aligned}
 P(X_{ij1}) &= P(X_{ij} = 1) \\
 &\dots \\
 P(X_{ijk}) &= \frac{P(X_{ij} = k)}{\prod_{l=1}^{k-1} (1 - P(X_{ijl}))}
 \end{aligned}$$

PITA uses tabling, see Section 1.4.2, that ensures that, when a goal is asked again, the answers for it already computed are retrieved rather than recomputed. This saves time because explanations for different goals are factored as in PRISM. Moreover, it also avoids non-termination in many cases.

PITA also exploits XSB Prolog's *answer subsumption* feature [Swift and Warren, 2012] that, when a new answer for a tabled subgoal is found, combines old answers with the new one according to a partial order or lattice.

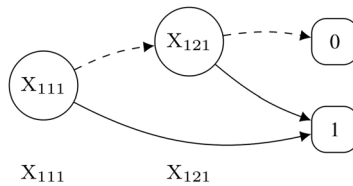


Figure 5.7 BDD representing the function in Equation (5.2).

For example, if the lattice is on the second argument of a binary predicate p , answer subsumption may be specified by means of the declaration

```
:-table p(_, lattice(or/3)).
```

where $or/3$ is the join operation of the lattice. Thus, if a table has an answer $p(a, d1)$ and a new answer $p(a, d2)$ is derived, the answer $p(a, d1)$ is replaced by $p(a, d3)$, where $d3$ is obtained by calling $or(d1, d2, d3)$.

In PITA, various predicates should be declared as tabled. For a predicate p/n , the declaration is

```
:-table p(_1, ..., _n, lattice(or/3)).
```

which indicates that answer subsumption is used to form the disjunction of BDDs in the last argument.

At a minimum, the predicate of the goal and all the predicates appearing in negative literals should be tabled with answer subsumption.

Thanks to answer subsumption, a call to a goal $PITA(a, D)$ where b is ground returns in D a BDD encoding the set of all of its explanations. Therefore, the transformation $PITA(b, D)$ for a negative literal $b = \sim a$ in the body of a rule first calls $PITA(a, DN)$. If the call fails, a does not have any explanation, so the BDD for b should be the one Boolean function. Otherwise, the BDD DN is negated with $not/2$. So the transformation of the body of a clause makes sure that, to compute the BDD associated with the head atom, the BDDs of all literals are computed and conjoined with the BDD encoding the choice relative to the clause.

If predicates appearing in negative literals are not tabled with answer subsumption, PITA is not correct as a call to a subgoal does not collect all explanations. It is usually useful to table every predicate whose answers have multiple explanations and are going to be reused often since in this way repeated computations are avoided and explanations are factored.

PITA was originally available only for XSB because it was the only Prolog offering answer subsumption. Recently, answer subsumption was included in SWI-Prolog and PITA is now available also for it in the `cpint` suite for SWI-Prolog. In Prologs without answer subsumption, such as YAP, answer subsumption can be simulated using a slightly different PITA transformation with explicit *findall/3* calls. *findall(Template, Goal, Bag)* creates the list *Bag* of instantiations of *Template* for which *Goal* succeeds. For example, *findall(X, p(X), Bag)* returns in *Bag* the list of values of X for which $P(X)$ succeeds.

PITA was shown correct and terminating under mild conditions [Riguzzi and Swift, 2013].

5.7 ProbLog2

The ProbLog2 system [Fierens et al., 2015] is a new version of ProbLog1 (Section 5.3) that performs knowledge compilation to Deterministic Decomposable Negation Normal Forms (d-DNNFs) rather than to BDDs.

ProbLog2 can perform the tasks CONDATOMS, EVID, and MPE over ProbLog programs. It also allows probabilistic intensional facts of the form

$$\Pi :: f(X_1, X_2, \dots, X_n) \leftarrow Body$$

with *Body* a conjunction of calls to non-probabilistic facts that define the domains of the variables X_1, X_2, \dots, X_n . Moreover, ProbLog2 allows annotated disjunctions in LPAD style of the form

$$\Pi_{i1} :: h_{i1} ; \dots ; \Pi_{in_i} :: h_{in_i} \leftarrow b_{i1}, \dots, b_{im_i}$$

which are equivalent to an LPAD clauses of the form

$$h_{i1} : \Pi_{i1} ; \dots ; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \dots, b_{im_i}$$

and are handled by translating them into probabilistic facts using the technique of Section 2.4. Let us call ProbLog2 this extension of the language of ProbLog.

Example 69 (Alarm – ProbLog2 [Fierens et al., 2015]). *The following program is similar to the alarm BN of Examples 10 and 27:*

```
0.1 :: burglary.
0.2 :: earthquake.
0.7 :: hears_alarm(X) ← person(X).
alarm ← burglary.
alarm ← earthquake.
calls(X) ← alarm, hears_alarm(X).
person(mary).
person(john).
```

Differently from the alarm BN of Examples 10 and 27, two people can call in case they hear the alarm.

ProbLog2 converts the program into a weighted Boolean formula and then performs WMC. A *weighted Boolean formula* is a Boolean formula

over a set of variables $\mathbf{V} = \{V_1, \dots, V_n\}$ associated with a weight function $w(\cdot)$ that assign a real number to each literal built on \mathbf{V} . The weight function is extended to assign a real number to each assignment $\omega = \{V_1 = v_1, \dots, V_n = v_n\}$ to the variables of \mathbf{V} :

$$w(\omega) = \prod_{l \in \omega} w(l)$$

where each variable assignment in ω is interpreted as a literal. Given the weighted Boolean formula ϕ , the *weighted model count* of ϕ , $WMC_{\mathbf{V}}(\phi)$, with respect to the set of variables \mathbf{V} , is defined as

$$WMC_{\mathbf{V}}(\phi) = \sum_{\omega \in SAT(\phi)} w(\omega).$$

where $SAT(\phi)$ is the set of assignments satisfying ϕ .

ProbLog2 converts the program into a weighted formula in three steps:

1. Grounding \mathcal{P} yielding a program \mathcal{P}_g , taking into account q and e in order to consider only the part of the program that is relevant to the query given the evidence.
2. Converting the ground rules in \mathcal{P}_g to an equivalent Boolean formula ϕ_r .
3. Taking into account the evidence and defining a weight function. A Boolean formula ϕ_e representing the evidence is conjoined with ϕ_r obtaining formula ϕ and a weight function is defined for all atoms in ϕ .

The grounding of the program can be restricted to the relevant rules only (see Section 1.3). SLD resolution is used to find relevant rules by proving all atoms in q, e . Tabling is used to avoid proving the same atom twice and to avoid going into infinite loops if the rules are cyclic. As programs are range-restricted, all the atoms in the rules used during the SLD resolution will eventually become ground, and hence also the rules themselves.

Moreover, inactive rules encountered during SLD resolution are omitted. A ground rule is *inactive* if the body of the rule contains a literal l that is false in the evidence (l can be an atom that is false in e , or the negation of an atom that is true in e). Inactive rules do not contribute to the probability of the query, so they can be safely omitted.

The relevant ground program contains all the information necessary for solving the corresponding EVID, COND, or MPE task.

Example 70 (Alarm – grounding – ProbLog2 [Fierens et al., 2015]). *If $q = \{\text{burglary}\}$ and $e = \text{calls}(\text{john})$ in Example 69, the relevant ground program is*

```
0.1 :: burglary.
0.2 :: earthquake.
0.7 :: hears_alarm(john).
alarm ← burglary.
alarm ← earthquake.
calls(john) ← alarm, hears_alarm(john).
```

The relevant ground program is now converted to an equivalent Boolean formula. The conversion is not merely syntactical as logic programming makes the Closed World Assumption while first-order logic doesn't.

If rules are acyclic, Clark's completion (see Section 1.4.1) can be used [Lloyd, 1987]. If rules are cyclic, i.e., they contain atoms that depend positively on each other, Clark's completion is not correct [Janhunen, 2004]. Two algorithms can then be used to perform the translation. The first [Janhunen, 2004] removes positive loops by introducing auxiliary atoms and rules and then applies Clark's completion. The second [Mantadelis and Janssens, 2010] first uses tabled SLD resolution to construct the proofs of all atoms in $\text{atoms}(q) \cup \text{atoms}(e)$, then collects the proofs in a data structure (a set of nested tries), and breaks the loops to build the Boolean formula.

Example 71 (Smokers – ProbLog [Fierens et al., 2015]). *The following program models causes for people to smoke: either they spontaneously start because of stress or they are influenced by one of their friends:*

```
0.2 :: stress(P) : ¬person(P).
0.3 :: influences(P1, P2) : ¬friend(P1, P2).
person(p1).
person(p2).
person(p3).
friend(p1, p2).
friend(p2, p1).
friend(p1, p3).
smokes(X) : ¬stress(X).
smokes(X) : ¬smokes(Y), influences(Y, X).
```

With the evidence $\text{smokes}(p2)$ and the query $\text{smokes}(p1)$, we obtain the following ground program:

0.2 :: $stress(p1)$.
 0.2 :: $stress(p2)$.
 0.3 :: $influences(p2, p1)$.
 0.3 :: $influences(p1, p2)$.
 $smokes(p1) : \neg stress(p1)$.
 $smokes(p1) : \neg smokes(p2), influences(p2, p1)$.
 $smokes(p2) : \neg stress(p2)$.
 $smokes(p2) : \neg smokes(p1), influences(p1, p2)$.

Clark's completion would generate the Boolean formula

$smokes(p1) \leftrightarrow stress(p1) \vee smokes(p2), influences(p2, p1)$.
 $smokes(p2) \leftrightarrow stress(p2) \vee smokes(p1), influences(p1, p2)$.

which has a model

$$\{smokes(p1), smokes(p2), \neg stress(p1), \neg stress(p2), \\ influences(p1, p2), influences(p2, p1), \dots\}$$

which is not a model of any world of the ground ProbLog program: for total choice

$$\{\neg stress(p1), \neg stress(p2), influences(p1, p2), influences(p2, p1)\}$$

the model assigns false to both $smokes(p1)$ and $smokes(p2)$.

The conversion algorithm of [Mantadelis and Janssens, 2010] generates:

$smokes(p1) \leftrightarrow aux1 \vee stress(p2)$
 $smokes(p2) \leftrightarrow aux2 \vee stress(p1)$
 $aux1 \leftrightarrow smokes(p2) \wedge influences(p2, p1)$
 $aux2 \leftrightarrow stress(p1) \wedge influences(p1, p2)$

Note that the loop in the original ProbLog program between $smokes(p1)$ and $smokes(p2)$ has been broken by using $stress(p1)$ instead of $smokes(p1)$ in the last formula.

Lemma 12 (Correctness of the ProbLog Program Transformation). *Let \mathcal{P}_g be a ground ProbLog. $SAT(\phi_r) = MOD(\mathcal{P}_g)$ where $MOD(\mathcal{P}_g)$ is the set of models of instances of \mathcal{P}_g .*

The final Boolean formula ϕ is built from the one for the rules, ϕ_r , and that for the evidence ϕ_e obtained as

$$\phi_e = \bigwedge_{\sim a \in e} \neg a \wedge \bigwedge_{a \in e} a$$

Then $\phi = \phi_r \wedge \phi_e$. We illustrate this on the Alarm example, which is acyclic.

Example 72 (Alarm – Boolean formula – ProbLog2 [Fierens et al., 2015]).

For Example 70, the Boolean formula ϕ is

$$\begin{aligned} \text{alarm} &\leftrightarrow \text{burglary} \vee \text{earthquake} \\ \text{calls}(\text{john}) &\leftrightarrow \text{alarm} \wedge \text{hears_alarm}(\text{john}) \\ \text{calls}(\text{john}) & \end{aligned}$$

Then the weight function $w(\cdot)$ is defined as: for each probabilistic fact $\Pi :: f$, f is assigned weight Π and $\neg f$ is assigned weight $1 - \Pi$. All the other literals are assigned weight 1. The weight of a world ω is given by the product of the weight of all literals in ω . The following theorem establishes the relationship between the relevant ground program and the weighted formula.

Theorem 13 (Model and weight equivalence [Fierens et al., 2015]). *Let \mathcal{P}_g be the relevant ground program for some ProbLog program with respect to q and e . Let $MOD_e(\mathcal{P}_g)$ be those models in $MOD(\mathcal{P}_g)$ that are consistent with the evidence e . Let ϕ denote the formula and $w(\cdot)$ the weight function of the weighted formula derived from \mathcal{P}_g . Then:*

- (model equivalence) $SAT(\phi) = MOD_e(\mathcal{P}_g)$,
- (weight equivalence) $\forall \omega \in SAT(\phi) : w(\omega) = P_{\mathcal{P}_g}(\omega)$, i.e., the weight of ω according to $w(\cdot)$ is equal to the probability of ω according to \mathcal{P}_g .

If \mathbf{V} is the set of the variables associated with $\mathcal{B}_{\mathcal{P}}$, then $WMC_{\mathbf{V}}(\phi) = P(e)$. When \mathbf{V} is clear from the context, it is omitted. So

$$P(e) = \sum_{\omega \in SAT(\phi)} \prod_{l \in \omega} w(l)$$

The inference tasks of COND, MPE, and EVID can be solved using state-of-the-art algorithms for WMC or weighted MAX-SAT.

By knowledge compilation, ProbLog2 translates ϕ to a smooth d-DNNF Boolean formula that allows WMC in polynomial time by in turn converting the d-DNNF into an arithmetic circuit. A Negation Normal Form (NNF) formula is a rooted directed acyclic graph in which each leaf node is labeled with a literal and each internal node is labeled with a conjunction or disjunction. Smooth d-DNNF formulas also satisfy

- Decomposability (D): for every conjunction node, no couple of children of the node has any variable in common.
- Determinism (d): for every disjunction node, every couple of children represents formulas that are logically inconsistent with each other.

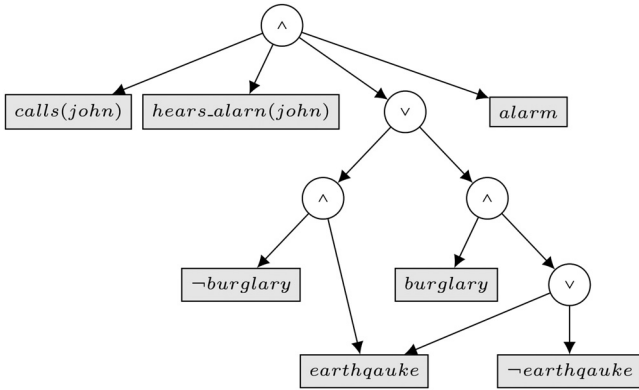


Figure 5.8 d-DNNF for the formula of Example 72. From [Fierens et al., 2015].

- **Smoothness:** for every disjunction node, all children use exactly the same set of variables.

Compilers for d-DNNF usually start from formulas in Conjunctive Normal Form (CNF). A CNF is a Boolean formula that takes the form of a conjunction of disjunctions of literals, i.e., a formula of the form:

$$l_{11} \vee \dots \vee l_{1m_1} \wedge \dots \wedge l_{n1} \vee \dots \vee l_{nm_n}$$

where each l_{ij} is a literal. Examples of compilers from CNF to d-DNNF are c2d Darwiche [2004] and DSHARP Muise et al. [2012]. The formula of Example 72 is translated to the d-DNNF of Figure 5.8.

The conversion of a d-DNNF formula into an arithmetic circuit is done in two steps [Darwiche, 2009, Chapter 12]: first conjunctions are replaced by multiplications and disjunctions by summations, and then each leaf node labeled with a literal l is replaced by a subtree consisting of a multiplication node with two children, a leaf node with a Boolean indicator variable $\lambda(l)$ for the literal l and a leaf node with the weight of l . The circuit for the Alarm example is shown in Figure 5.9. This transformation is equivalent to transforming the WMC formula into

$$WMC(\phi) = \sum_{\omega \in SAT(\phi)} \prod_{l \in \omega} w(l)\lambda(l) = \sum_{\omega \in SAT(\phi)} \prod_{l \in \omega} w(l) \prod_{l \in \omega} \lambda(l)$$

Given the arithmetic circuit, the WMC can be computed by evaluating the circuit bottom-up after having assigned the value 1 to all the indicator variables. The value computed for the root, f , is the probability of evidence and

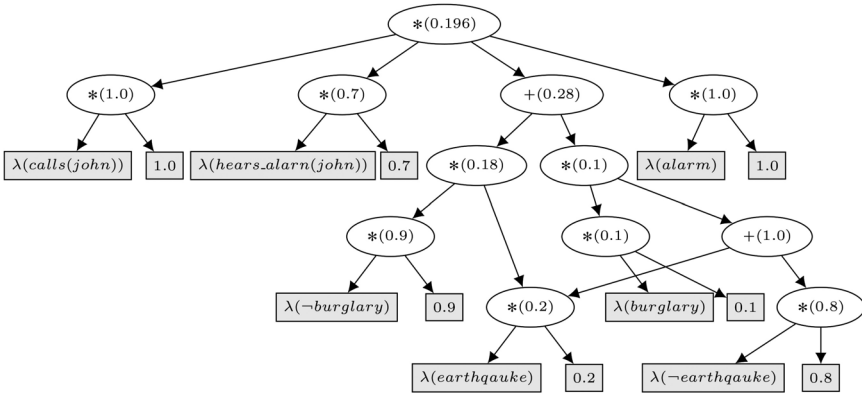


Figure 5.9 Arithmetic circuit for the d-DNNF of Figure 5.8. From [Fierens et al., 2015].

so solves EVID. Figure 5.9 shows the values that are computed for each node: the value for the root, 0.196, is the probability of evidence.

With the arithmetic circuit, it is also possible to compute the probability of any evidence, provided that it extends the initial evidence. To compute $P(e, l_1 \dots l_n)$ for any conjunction of literals l_1, \dots, l_n , it is enough to set the indicator variables as $\lambda(l_i) = 1$, $\lambda(\neg l_i) = 0$ (where $\neg \neg a = a$) and $\lambda(l) = 1$ for the other literals l , and evaluate the circuit. In fact, the value $f(l_1 \dots l_n)$ of the root node will give:

$$\begin{aligned}
 f(l_1 \dots l_n) &= \sum_{\omega \in \text{SAT}(\phi)} \prod_{l \in \omega} w(l) \prod_{l \in \omega} \begin{cases} 1, & \text{if } \{l_1 \dots l_n\} \subseteq \omega \\ 0, & \text{otherwise} \end{cases} = \\
 &= \sum_{\omega \in \text{SAT}(\phi), \{l_1 \dots l_n\} \subseteq \omega} \prod_{l \in \omega} w(l) = \\
 &= P(e, l_1 \dots l_n)
 \end{aligned}$$

So in theory, one could build the circuit for formula ϕ_r only, since the probability of any set of evidence can then be computed. The formula for evidence however usually simplifies the compilation process and the resulting circuit.

To compute CONDATOMS, we are given a set of atoms Q and evidence e and we want to compute $P(q|e)$ for all atoms $q \in Q$. Given the definition of conditional probability, $P(q|e) = \frac{P(q,e)}{P(e)}$, COND could be solved by computing the probability of evidence q, e for all atoms in Q . However, consider the partial derivative $\frac{\partial f}{\partial \lambda_q}$ for an atom q :

$$\begin{aligned} \frac{\partial f}{\partial \lambda_q} &= \sum_{\omega \in SAT(\phi), q \in \omega} \prod_{l \in \omega} w(l) \prod_{l \in \omega, l \neq q} \lambda(l) = \\ &= \sum_{\omega \in SAT(\phi), q \in \omega} \prod_{l \in \omega} w(l) = \\ &P(e, q) \end{aligned}$$

So if we compute the partial derivatives of f for all indicator variables $\lambda(q)$, we get $P(q, e)$ for all atoms q . We can solve this problem by traversing the circuit twice, once bottom-up and once top-down; see [Darwiche, 2009, Algorithms 34 and 35]. The algorithm computes the value $v(n)$ of each node n and the derivative $d(n)$ of the value of the root node r with respect to n , i.e., $d(n) = \frac{\partial v(r)}{\partial v(n)}$. $v(n)$ is computed by traversing the circuit bottom-up and evaluating it at each node. $d(n)$ can be computed by observing that $d(r) = 1$ and, by the chain rule of calculus, for an arbitrary non-root node n with p indicating its parents

$$d(n) = \sum_p \frac{\partial v(r)}{\partial v(p)} \frac{\partial v(p)}{\partial v(n)} = \sum_p d(p) \frac{\partial v(p)}{\partial v(n)}.$$

If parent p is a multiplication node with n' indicating its children

$$\frac{\partial v(p)}{\partial v(n)} = \frac{\partial v(n) \prod_{n' \neq n} v(n')}{\partial v(n)} = \prod_{n' \neq n} v(n').$$

If parent p is an addition node with n' indicating its children

$$\frac{\partial v(p)}{\partial v(n)} = \frac{\partial v(n) + \sum_{n' \neq n} v(n')}{\partial v(n)} = 1.$$

So, if we indicate with $+p$ an addition parent of n and with $*p$ a multiplication parent of n , then

$$d(n) = \sum_{+p} d(+p) + \sum_{*p} d(*p) \prod_{n' \neq n} v(n').$$

Moreover, $\frac{\partial v(*p)}{\partial v(n)}$ can be computed as $\frac{\partial v(*p)}{\partial v(n)} = \frac{v(*p)}{v(n)}$ if $v(n) \neq 0$. If all indicator variables are set to 1, as required to compute f , and if no parameter is 0, which can be assumed as otherwise the formula could be simplified, then $v(n) \neq 0$ for all nodes and

$$d(n) = \sum_{+p} d(+p) + \sum_{*p} d(*p) v(*p) / v(n).$$

This leads to Procedure CIRCP shown in Algorithm 5 that is a simplified version of [Darwiche, 2009, Algorithm 35] for the case $v(n) \neq 0$ for all nodes. $v(n)$ may be 0 if f is evaluated at additional evidence (see $f(l_1 \dots l_n)$ above), in that case, [Darwiche, 2009, Algorithm 35] must be used that takes into account this case and is slightly more complex.

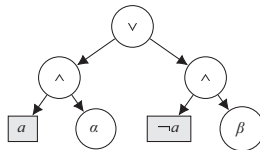
Algorithm 5 Procedure CIRCP: Computation of value and derivatives of circuit nodes.

```

1: procedure CIRCP(circuit)
2:   assign values to leaves
3:   for all non-leaf node  $n$  with children  $c$  (visit children before parents) do
4:     if  $n$  is an addition node then
5:        $v(n) \leftarrow \sum_c v(c)$ 
6:     else
7:        $v(n) \leftarrow \prod_c v(c)$ 
8:     end if
9:   end for
10:   $d(r) \leftarrow 1, d(n) = 0$  for all non-root nodes
11:  for all non-root node  $n$  (visit parents before children) do
12:    for all parents  $p$  of  $n$  do
13:      if  $p$  is an addition parent then
14:         $d(n) = d(n) + d(p)$ 
15:      else
16:         $d(n) \leftarrow d(n) + d(p)v(p)/v(n)$ 
17:      end if
18:    end for
19:  end for
20: end procedure

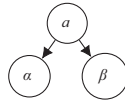
```

ProbLog2 also offers compilation to BDDs. In this case, EVID and COND can be performed with Algorithm 4. In fact, Algorithm 4 performs WMC over BDDs. This can be seen by observing that BDDs are d-DNNF that also satisfy the properties of decision and ordering [Darwiche, 2004]. A d-DNNF satisfies the property of *decision* iff the root node is a decision node, i.e., a node labeled with 0, 1 or the subtree



where a is a variable and α and β are decision nodes. a is called the *decision variable*. A d-DNNF satisfying the property of decision also satisfies the property of *ordering* if decision variables appear in the same order along any

path from the root to any leaf. A d-DNNF satisfying decision and ordering is a BDD. d-DNNF satisfying decision and ordering may seem different from a BDD as we have seen it: however, if each decision node of the form above is replaced by



we obtain a BDD.

The d-DNNF of Figure 5.8, for example, does not satisfy decision and ordering. The same Boolean formula, however, can be encoded with the BDD of Figure 5.10 that can be converted to a d-DNNF using the above equivalence. Algorithm 4 for computing the probability of a BDD is equivalent to the evaluation of the arithmetic circuit that can be obtained from the BDD by seeing it as a d-DNNF.

More recently, ProbLog2 has also included the possibility of compiling the Boolean function to Sentential Decision Diagrams (SDDs)

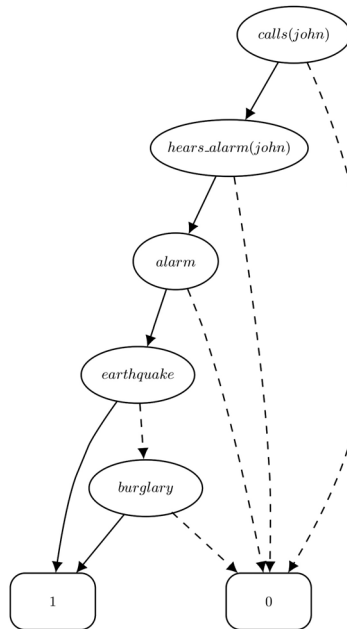


Figure 5.10 BDD for the formula of Example 72.

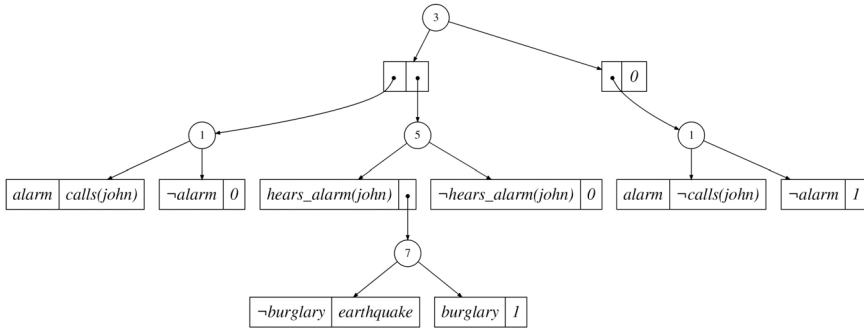


Figure 5.11 SDD for the formula of Example 72.

[Vlasselaer et al., 2014; Dries et al., 2015]. An SDD for the formula of Example 72 is shown in Figure 5.11.

An SDD [Darwiche, 2011] contains two types of nodes: *decision nodes*, represented as circles, and *elements*, represented as paired boxes. Elements are the children of decision nodes and each box in an element can contain a pointer to a decision node or a *terminal node*, either a literal or the constants 0 or 1. In an element (p, s) , p is called a *prime* and s is called a *sub*. A decision node with children $(p_1, s_1), \dots, (p_n, s_n)$ represents the function $(p_1 \wedge s_1) \vee \dots \vee (p_n \wedge s_n)$. Primes p_1, \dots, p_n must form a partition: $p_i \neq 0, p_i \wedge p_j = 0$ for $i \neq j$, and $p_1 \vee \dots \vee p_n = 1$.

A *vtree* is a full binary tree whose leaves are in one-to-one correspondence with the formula variables. Each SDD is normalized for some vtree. Figure 5.12 shows the vtree for which the SDD of Figure 5.11 is normalized. Each SDD node is normalized for some vtree node. The root node of the SDD is normalized for the root vtree node. Terminal nodes are normalized for leaf vtree nodes. If a decision node is normalized for a vtree node v , then

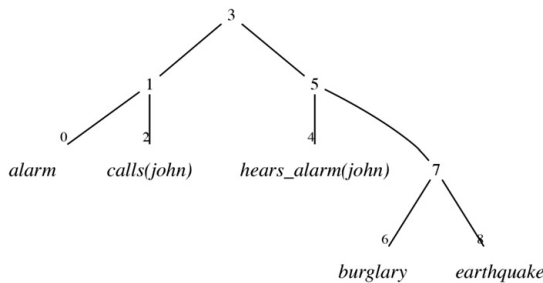


Figure 5.12 vtree for which the SDD of Figure 5.11 is normalized.

its primes are normalized for the left child of v , and its subs are normalized for the right child of v . As a result, primes and subs of same decision node do not share variables. The SDD of a Boolean formula is unique once a vtree is fixed. In Figure 5.11, decision nodes are labeled with the vtree nodes they are normalized for.

SDDs are special cases of d-DNNF: if one replaces circle-nodes with or-nodes, and paired-boxes with and-nodes, one obtains a d-DNNF. SDDs satisfy two additional properties with respect to d-DNNF: structured decomposability and strong determinism.

To define structured decomposability, consider a d-DNNF δ and assume, without loss of generality, that all conjunctions are binary. δ *respects* a vtree V if for every conjunction $\alpha \wedge \beta$ in δ , there is a node v in V such that $\text{vars}(\alpha) \subseteq \text{vars}(v_l)$ and $\text{vars}(\beta) \subseteq \text{vars}(v_r)$ where v_l and v_r are the left and right child of v and $\text{vars}(\alpha)$ and $\text{vars}(v)$ is the set of variables appearing in the subdiagram rooted at α and the sub-vtree rooted at v respectively. δ enjoys *structured decomposability* if it satisfies some vtree.

Strong determinism requires not only that the children of or nodes are pairwise inconsistent but that they form a strongly deterministic decomposition. An (\mathbf{X}, \mathbf{Y}) -*decomposition* of a function $f(\mathbf{X}, \mathbf{Y})$ over non-overlapping variables \mathbf{X} and \mathbf{Y} is a set $\{(p_1, s_1), \dots, (p_n, s_n)\}$ such that

$$f = p_1(\mathbf{X}) \wedge s_1(\mathbf{Y}) \vee \dots \vee p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})$$

If $p_i \wedge p_j = 0$ for $i \neq j$, the decomposition is said to be *strongly deterministic*. A d-DNNF is *strongly deterministic* if each or node is a strongly deterministic decomposition.

BDDs are a special case of SDDs where decompositions are all Shannon: formula f is decomposed into $\{(X, f^X), (\neg X, f^{\neg X})\}$. SDDs generalize BDDs by considering non-binary decisions based on the value of primes and by considering vtrees instead of linear variable orders.

In ProbLog2, the user can choose whether to use d-DNNFs or SDDs. The choice of the compilation language depends on the tradeoff between succinctness and tractability. Succinctness is defined by the size of a knowledge base once it is compiled. Tractability is determined by the set of operations that can be performed in polynomial time. The more tractable a representation is, the less succinct it is.

A language is *at least as succinct as* another if, for every sentence in the second language, there exists an equivalent sentence in the first with a polynomially smaller size. The succinctness order for BDD, SDD, and d-DNNF is

$$\text{d-DNNF} < \text{SDD} \leq \text{BDD}$$

Table 5.1 Tractability of operations. ? means “unknown”, ✓ means “tractable” and ○ means “not tractable unless P=NP” Operations are meant over a bounded number of operands and BDDs operands should have the same variable order and SDDs the same vtree. From [Vlasselaer et al., 2014]

Language	Negation	Conjunction	Disjunction	Model Counting
d-DNNF	?	○	○	✓
SDD	✓	✓	✓	✓
BDD	✓	✓	✓	✓

meaning that d-DNNF is strictly more succinct than SDDs and that SDDs are at least as succinct as BDDs (whether $SDD < BDD$ is an open problem). Since $SDD \not\subseteq d\text{-DNNF}$, there exist formulas whose smallest SDD representation is exponentially larger than its d-DNNF representation.

The operations for which we consider tractability are those that are useful for probabilistic inference, namely, negation, conjunction, disjunction, and model counting. The situation for the languages we consider is summarized in Table 5.1.

SDDs and BDDs support tractable Boolean combination operators. This means that they can be built bottom-up starting from the cycle-free ground program, similarly to what is done by ProbLog1 or PITA. d-DNNF compilers instead require the formula to be in CNF. Converting the Clark’s completion of the rules into CNF introduces a new auxiliary variable for each rule which has a body with more than one literal. This adds cost to the compilation process.

Another advantage of BDDs and SDDs is that they support minimization: their size can be reduced by modifying the variable order or vtree. Minimization of d-DNNF is not supported, so circuits may be larger than necessary.

The disadvantage of BDDs with respect to d-DNNF and SDDs is that they have worse size upper bounds, while the upper bound for d-DNNF and SDD is the same [Razgon, 2014].

Experimental comparisons confirm this and show that d-DNNF leads to faster inference than BDDs [Fierens et al., 2015] and that SDDs lead to faster inference than d-DNNF [Vlasselaer et al., 2014].

The ProbLog2 system can be used in three ways. It can be used online at <https://dtai.cs.kuleuven.be/problog/>: the user can enter and solve ProbLog problems with just a web browser.

It can be used from the command line as a Python program. In this case, the user has full control over the system settings and can use all machine

resources, in contrast with the online version that has resource limits. In this version, ProbLog programs may use external functions written in Python, such as those offered by the wide Python ecosystem.

ProbLog2 can also be used as a library that can be called from Python for building and querying probabilistic ProbLog models.

5.8 T_P Compilation

Previous inference algorithms work backward, by reasoning from the query toward the probabilistic choices. T_P compilation [Vlasselaer et al., 2015, 2016] is an approach for performing probabilistic inference in ProbLog using forward reasoning. In particular, it is based on the operator T_{CP} , a generalization of the T_P operator of logic programming that operates on parameterized interpretations. We have encountered a form of parameterized interpretations in Section 3.2 where each atom is associated with a set of composite choices. The parameterized interpretations of [Vlasselaer et al., 2016] associate ground atoms with Boolean formulas built on variables that represent probabilistic ground facts.

Definition 34 (Parameterized interpretation [Vlasselaer et al., 2015, 2016]). *A parameterized interpretation \mathcal{I} of a ground probabilistic logic program \mathcal{P} with probabilistic facts \mathcal{F} and atoms $\mathcal{B}_{\mathcal{P}}$ is a set of tuples (a, λ_a) with $a \in \mathcal{B}_{\mathcal{P}}$ and λ_a a propositional formula over \mathcal{F} expressing in which interpretations a is true.*

The T_{CP} operator takes as input a parameterized interpretation and returns another parameterized interpretation obtained by applying the rules once.

Definition 35 (T_{CP} operator [Vlasselaer et al., 2015, 2016]). *Let \mathcal{P} be a ground probabilistic logic program with probabilistic facts \mathcal{F} and atoms $\mathcal{B}_{\mathcal{P}}$. Let \mathcal{I} be a parameterized interpretation with pairs (a, λ_a) . Then, the T_{CP} operator is $T_{CP}(\mathcal{I}) = \{(a, \lambda_a) \mid a \in \mathcal{B}_{\mathcal{P}}\}$ where*

$$\lambda'_a = \begin{cases} a & \text{if } a \in \mathcal{F} \\ \bigvee_{a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, \sim c_m \in \mathcal{R}} (\lambda_{b_1} \wedge \dots \wedge \lambda_{b_n} \wedge \neg \lambda_{c_1} \wedge \dots \wedge \neg \lambda_{c_m}) & \text{if } a \in \mathcal{B}_{\mathcal{P}} \setminus \mathcal{F} \end{cases}$$

The ordinal powers of T_{CP} start from $\{(a, 0) \mid a \in \mathcal{B}_{\mathcal{P}}\}$. The concept of fixpoint must be defined in a semantic rather than syntactic way.

Definition 36 (Fixpoint of T_{CP} [Vlasselaer et al., 2015, 2016]). *A parameterized interpretation \mathcal{I} is a fixpoint of the T_{CP} operator if and only if for all $a \in \mathcal{B}_{\mathcal{P}}$, $\lambda_a \equiv \lambda'_a$, where λ_a and λ'_a are the formulas for a in \mathcal{I} and $T_{CP}(\mathcal{I})$, respectively.*

Vlasselaer et al. [2016] show that, for definite programs, T_{CP} has a least fixpoint $\text{lfp}(T_{CP}) = \{(a, \lambda_a^\infty) \mid a \in \mathcal{B}_{\mathcal{P}}\}$ where the λ_a^∞ s exactly describe the possible worlds where a is true and can be used to compute the probability for each atom by WMC, i.e., $P(a) = \text{WMC}(\lambda_a^\infty)$.

The probability of an atomic query q given evidence e can be computed as

$$P(q|e) = \frac{\text{WMC}(\lambda_q \wedge \lambda_e)}{\text{WMC}(\lambda_e)}$$

where $\lambda_e = \bigwedge_{e_i \in e} \lambda_{e_i}$.

The approach can also be applied to stratified normal logic programs by iterating the T_{CP} operator stratum by stratum: the fixpoint of T_{CP} is computed by considering the rules for each stratum in turn.

So, to perform exact inference on a ProbLog program \mathcal{P} , the T_{CP} operator should be iterated stratum by stratum until the fixpoint is reached in each stratum. The parameterized interpretation that is obtained after the last stratum can then be used to perform COND and EVID.

The algorithm of [Vlasselaer et al., 2016] represents the formulas in the interpretations by means of SDDs. So the Boolean formulas λ_a in the definition of T_{CP} are replaced by SDD structures Λ_a . Since negation, conjunction, and disjunction are efficient for SDDs, then so is the application of T_{CP} .

Moreover, the T_{CP} operator can be applied in a granular way one atom at a time, which is useful for selecting more effective evaluation strategies in approximate inference, see Section 7.9. The operator $T_{CP}(a, \mathcal{I})$ considers only the rules with a in the head and updates only the formula λ_a . So an application of T_{CP} is modified in

1. select an atom $a \in \mathcal{B}_{\mathcal{P}}$,
2. compute $T_{CP}(a, \mathcal{I})$.

Vlasselaer et al. [2016] show that if each atom is selected frequently enough in step 1, then the same fixpoint $\text{lfp}(T_{CP})$ is reached as for the naive algorithm, provided that the operator is still applied stratum by stratum in normal logic programs.

T_P compilation can also be used for performing inference in the case of updates of the program, where (ground) facts and rules can be added or

removed. For definite programs, past compilation results can be reused to compute the new fixpoint. Moreover, T_P compilation can be used for dynamic models, where each atom has an argument that represents the time at which it is true. In this case, rules express the dependency of an atom at a time step from atoms at the same or previous time step. An extension of T_P compilation can then be used for filtering, i.e., computing the probability of a query at a time t given evidence up to time t .

Experiments in [Vlasselaer et al., 2016] show that T_P compilation compares favorably with ProbLog2 with both d-DNNF and SDD.

5.9 Modeling Assumptions in PITA

Let us recall here PRISM modeling assumptions:

1. The probability of a conjunction (A, B) is computed as the product of the probabilities of A and B (*independent-and assumption*).
2. The probability of a disjunction $(A; B)$ is computed as the sum of the probabilities of A and B (*exclusive-or assumption*).

These assumptions can be stated more formally by referring to explanations. Given an explanation κ , let $RV(\kappa) = \{C_i\theta_j | (C_i, \theta_j, k) \in \kappa\}$. Given a set of explanations K , let $RV(K) = \bigcup_{\kappa \in K} RV(\kappa)$. Two sets of explanations, K_1 and K_2 , are *independent* if $RV(K_1) \cap RV(K_2) = \emptyset$ and *exclusive* if, $\forall \kappa_1 \in K_1, \kappa_2 \in K_2, \kappa_1$ and κ_2 are incompatible.

The independent-and assumption means that, when deriving a covering set of explanations for a goal, the covering sets of explanations K_i and K_j for two ground subgoals in the body of a clause are independent.

The exclusive-or assumption means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are exclusive. This implies that the atom h is derived using clauses that have mutually exclusive bodies, i.e., that their bodies are not true at the same time in any world.

The systems PRISM [Sato and Kameya, 1997] and PITA(IND, EXC) [Riguzzi and Swift, 2011] exploit these assumptions to speed up the computation. In fact, these assumptions make the computation of probabilities “truth-functional” [Gerla, 2001] (the probability of conjunction/disjunction of two propositions depends only on the probabilities of those propositions), while in the general case, this is false. PITA(IND, EXC) differs from PITA in the definition of the *one/1*, *zero/1*, *not/2*, *and/3*, *or/3*, and *equality/4*

predicates that now work on probabilities P rather than on BDDs. Their definitions are

$zero(0)$.
 $one(1)$.
 $not(A, B) \leftarrow B \text{ is } 1 - A$.
 $and(A, B, C) \leftarrow C \text{ is } A * B$.
 $or(A, B, C) \leftarrow C \text{ is } A + B$.
 $equality(V, _N, P, P)$.

Instead of the *exclusive-or assumption*, a program may satisfy the following assumption:

3. The probability of a disjunction $(A; B)$ is computed as if A and B were independent (*independent-or assumption*).

This means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are independent. If A and B are independent, the probability of their disjunction is

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B) = \\ P(A) + P(B) - P(A)P(B)$$

by the laws of probability theory. PITA(IND, EXC) can be used for programs respecting this assumption by changing the *or/3* predicate in this way

$or(A, B, P) \leftarrow P \text{ is } A + B - A * B$.

PITA(IND,IND) is the resulting system.

The exclusiveness assumption for conjunctions of literals means that the conjunction is true in 0 worlds and thus has probability 0, so it does not make sense to consider a PITA(EXC,_) system.

The following program

$path(Node, Node)$.
 $path(Source, Target) : 0.3 \leftarrow edge(Source, Node),$
 $path(Node, Target)$.
 $edge(0, 1) : 0.3$.

...

satisfies the independent-and and independent-or assumptions depending on the structure of the graph. For example, the graphs in Figures 5.13(a) and 5.13(b) respect these assumptions for the query $path(0, 1)$. Similar graphs of increasing sizes can be obtained [Bragaglia and Riguzzi, 2011]. We call the first graph type a “lanes” graph and the second a “branches” graph. The graphs of the type of Figure 5.13(c), called “parachutes” graphs, instead, satisfy only the independent-and assumption for the query $path(0, 1)$.

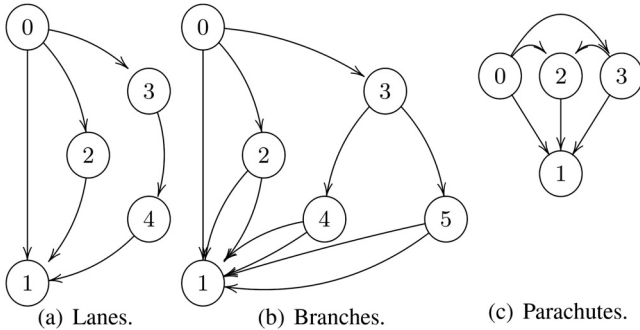


Figure 5.13 Examples of graphs satisfying some of the assumptions. From [Bragaglia and Riguzzi, 2011].

All three types of graphs respect the independent-and assumption because, when deriving the goal $path(0, 1)$, paths are built incrementally starting from node 0 and adding one edge at a time with the second clause of the definition of $path/2$. Since the edge that is added does not appear in the rest of the path, the assumption is respected.

Lanes and branches graphs respect the independent-or assumption because, when deriving the goal $path(0, 1)$, ground instantiations of the second path clause have $path(i, 1)$ in the head and originate atomic choices of the form $(C_2, \{Source/i, Target/1, Node/j\}, 1)$.

Explanations for $path(i, 1)$ also contain atomic choices $(e_{i,j}, \emptyset, 1)$ for every fact $edge(i, j) : 0.3$. in the path. Each explanation corresponds to a path. In the lanes graph, each node except 0 and 1 lies on a single path, so the explanations for $path(i, 1)$ do not share random variables. In the branches graphs, each explanation for $path(i, 1)$ depends on a disjoint set of edges. In the parachutes graph, instead this is not true: for example, the path from 2 to 1 shares the edge from 2 to 1 with the path 3, 2, 1.

Another program satisfying the independent-and and independent-or assumptions is the following

```
sametitle(A, B) : 0.3 ←
    haswordtitle(A, word_10),
    haswordtitle(B, word_10).
sametitle(A, B) : 0.3 ←
    haswordtitle(A, word_1321),
    haswordtitle(B, word_1321).
```

...

which computes the probability that the titles of two different citations are the same on the basis of the words that are present in the titles. The dots

stand for clauses differing from the ones above only for the words considered. The *haswordtitle/2* predicate is defined by a set of certain facts. This is part of a program to disambiguate citations in the Cora database [Singla and Domingos, 2005]. The program satisfies the independent-and assumption for the query

$$\text{sametitle}(\text{tit1}, \text{tit2})$$

because

$$\text{haswordtitle}/2$$

has no uncertainty. It satisfies the independent-or assumption because each clause for *sametitle/2* defines a different random variable.

5.9.1 PITA(OPT)

PITA(OPT) [Riguzzi, 2014] differs from PITA because it checks for the truth of the assumptions before applying BDD logical operations. If the assumptions hold, then the simplified probability computations are used.

The data structures used to store probabilistic information in PITA(OPT) are couples (P, T) where P is a real number representing a probability and T is a term formed with the functors *zero/0*, *one/0*, *c/2*, *or/2*, *and/2*, *not/1*, and the integers. If T is an integer, it represents a pointer to the root node of a BDD. If T is not an integer, it represents a Boolean expression of which the terms of the form *zero*, *one*, *c(var, val)* and the integers represent the base case: *c(var, val)* indicates the equation $\text{var} = \text{val}$ while an integer indicates a BDD. In this way, we are able to represent Boolean formulas by means of either a BDD, a Prolog term, or a combination thereof.

For example, *or(0x94ba008, and(c(1, 1), not(c(2, 3)))* represents the expression: $B \vee (X_1 = 1 \wedge \neg(X_2 = 3))$ where B is the Boolean function represented by the BDD whose root node address in memory is the integer *0x94ba008* in Prolog hexadecimal notation.

PITA(OPT) differs from PITA also in the definition of *zero/1*, *one/1*, *not/2*, *and/3*, *or/3*, and *equality/4* that now work on couples (P, T) rather than on BDDs. *equality/4* is defined as

$$\text{equality}(V, N, P, (P, c(V, N))).$$

The *one/1* and *zero/1* predicates are defined as

$$\text{zero}((0, \text{zero})).$$

$$\text{one}((1, \text{one})).$$

The *or/3* and *and/3* predicates first check whether one of their input argument is (an integer pointing to) a BDD. If so, they also convert the other input argument to a BDD and test for independence using the library function

$bdd_ind(B1, B2, I)$. Such a function is implemented in C and uses the CUDD function `Cudd_SupportIndex` that returns an array indicating which variables appear in a BDD (the support variables). $bdd_ind(B1, B2, I)$ checks whether there is an intersection between the set of support variables of $B1$ and $B2$ and returns $I = 1$ if the intersection is empty. If the two BDDs are independent, then the value of the resulting probability is computed using a formula and a compound term is returned.

If none of the input arguments of $or/3$ and $and/3$ are BDDs, then these predicates test whether the independent-and or the exclusive-or assumptions hold. If so, they update the value of the probability using a formula and return a compound term. If not, they convert the terms to BDDs, apply the corresponding operation, and return the resulting BDD together with the probability it represents. The code for $or/3$ and $and/3$ is shown in Figures 5.14 and 5.15, respectively, where Boolean operation between BDDs are prefixed with $bdd_$.

```

or((PA,TA),(PB,TB),(PC,TC)) ←
  ((integer(TA);integer(TB)) →
   ev(TA,BA),ev(TB,BB),
   bdd_ind(BA,BB,I),
   (I = 1 →
    PC is PA + PB - PA * PB,
    TC = or(BA,BB)
   )
  ;
  bdd_or(BA,BB,TC),ret_prob(TC,PC)
 )
;
(ind(TA,TB) →
 PC is PA + PB - PA * PB,
 TC = or(BA,BB)
 ;
 (exc(TA,TB) →
  PC is PA + PB,
  TC = or(BA,BB)
 ;
  ev(TA,BA),ev(TB,BB),
  bdd_or(BA,BB,TC),ret_prob(TC,PC)
 )
 )
 ).

```

Figure 5.14 Code for the $or/3$ predicate of PITA(OPT).

```

and((PA,TA), (PB,TB), (PC,TC)) ←
  ((integer(TA); integer(TB)) →
    ev(TA, BA), ev(TB, BB),
    bdd_ind(A, BB, I),
    (I = 1 →
      PC is PA * PB,
      TC = and(BA, BB)
    )
  );
  bdd_and(BA, BB, TC), ret_prob(TC, PC)
  (bdd_zero(TC) →
    fail
  );
  true
)
)
);
(ind(TA, TB) →
  PC is PA * PB,
  TC = and(BA, BB)
);
(exc(TA, TB) →
  fail
);
  ev(TA, BA), ev(TB, BB),
  bdd_and(BA, BB, TC), ret_prob(TC, PC)
)
)
).

```

Figure 5.15 Code for the *and/3* predicate of PITA(OPT).

In these predicate definitions, *ev/2* evaluates a term returning a BDD. In *and/3*, after the first *bdd_and/3* operation, a test is made to check whether the resulting BDD represent the 0 constant. If so, the derivation fails as this branch contributes with a 0 probability. These predicates make sure that, once a BDD has been built, it is used in the following operations, avoiding the manipulation of terms and exploiting the work performed as much as possible.

The *not/2* predicate is very simple: it complements the probability and returns a new term:

$$\text{not}((P, B), (P1, \text{not}(B))) \leftarrow P1 \text{ is } 1 - P.$$

The predicate *exc/2* checks for the exclusiveness between two terms with a recursion through the structure of the terms, see Figure 5.16.

```

exc(zero, _) ←!.
exc(_, zero) ←!.
exc(c(V, N), c(V, N1)) ←!, N\ = N1.
exc(c(V, N), or(X, Y)) ←!, exc(c(V, N), X),
  exc(c(V, N), Y).
exc(c(V, N), and(X, Y)) ←!, (exc(c(V, N), X);
  exc(c(V, N), Y)).
exc(or(A, B), or(X, Y)) ←!, exc(A, X), exc(A, Y),
  exc(B, X), exc(B, Y).
exc(or(A, B), and(X, Y)) ←!, (exc(A, X); exc(A, Y)),
  (exc(B, X); exc(B, Y)).
exc(and(A, B), and(X, Y)) ←!, exc(A, X); exc(A, Y);
  exc(B, X); exc(B, Y).
exc(and(A, B), or(X, Y)) ←!, (exc(A, X); exc(B, X)),
  (exc(A, Y); exc(B, Y)).
exc(not(A), A) ←!.
exc(not(A), and(X, Y)) ←!, exc(not(A), X);
  exc(not(A), Y).
exc(not(A), or(X, Y)) ←!, exc(not(A), X),
  exc(not(A), Y).
exc(A, or(X, Y)) ←!, exc(A, X), exc(A, Y).
exc(A, and(X, Y)) ← exc(A, X); exc(A, Y).

```

Figure 5.16 Code for the *exc/2* predicate of PITA(OPT).

For example, the goal

$$\text{exc}(\text{or}(\text{c}(1, 1), \text{c}(2, 1)), \text{and}(\text{c}(1, 2), \text{c}(2, 2)))$$

matches the 7th clause and calls the subgoals

$$\text{exc}(\text{c}(1, 1), \text{c}(1, 2)), \text{exc}(\text{c}(1, 1), \text{c}(2, 2)), \text{exc}(\text{c}(2, 1), \text{c}(1, 2)), \\ \text{exc}(\text{c}(2, 1), \text{c}(2, 2)).$$

Of the first two calls, *exc*(*c*(1, 1), *c*(1, 2)) succeeds, thus satisfying the first conjunct in the body. Of the latter two calls, *exc*(*c*(2, 1), *c*(2, 2)) succeeds, thus satisfying the second conjunct in the body and proving the goal.

The *ind/2* predicate checks for independence between two terms. It visits the structure of the first term until it reaches an atomic choice. Then it checks for the absence of the variable in the second term with the predicate *absent/2*. The code for *ind/2* and *absent/2* is shown in Figure 5.17. For example, the goal

$$\text{ind}(\text{or}(\text{c}(1, 1), \text{c}(2, 1)), \text{and}(\text{c}(3, 2), \text{c}(4, 2)))$$

matches the 6th clause and calls

$$\text{ind}(\text{c}(1, 1), \text{and}(\text{c}(3, 2), \text{c}(4, 2))), \text{ind}(\text{c}(2, 1), \text{and}(\text{c}(3, 2), \text{c}(4, 2))).$$

```

ind(one, _) ←!.
ind(zero, _) ←!.
ind(_, one) ←!.
ind(_, zero) ←!.
ind(c(V, _N), B) ←!, absent(V, B).
ind(or(X, Y), B) ←!, ind(X, B), ind(Y, B).
ind(and(X, Y), B) ←!, ind(X, B), ind(Y, B).
ind(not(A), B) ← ind(A, B).
absent(V, c(V1, _N1)) ←!, V \ = V1.
absent(V, or(X, Y)) ←!, absent(V, X), absent(V, Y).
absent(V, and(X, Y)) ←!, absent(V, X), absent(V, Y).
absent(V, not(A)) ← absent(V, A).

```

Figure 5.17 Code for the *ind/2* predicate of PITA(OPT).

The first call matches the 5th clause and calls

$$\text{absent}(1, \text{and}(c(3, 2), c(4, 2)))$$

which, in turn, calls $\text{absent}(1, c(3, 2))$ and $\text{absent}(1, c(4, 2))$. Since they both succeed, $\text{ind}(c(1, 1), \text{and}(c(3, 2), c(4, 2)))$ succeeds as well. The second call matches the 5th clause and calls $\text{absent}(2, \text{and}(c(3, 2), c(4, 2)))$ which, in turn, calls $\text{absent}(2, c(3, 2))$ and $\text{absent}(2, c(4, 2))$. They both succeed so $\text{ind}(c(2, 1), \text{and}(c(3, 2), c(4, 2)))$ and the original goal is proved.

The predicates *exc/2* and *ind/2* define sufficient conditions for exclusion and independence, respectively. If the arguments of *exc/2* and *ind/2* do not contain integer terms representing BDDs, then the conditions are also necessary. The code for predicate *ev/2* for the evaluation of a term is shown in Figure 5.18.

When the program satisfies the (IND,EXC) or (IND,IND) assumptions, the PITA(OPT) algorithm answers the query without building BDDs: terms

```

ev(B, B) ← integer(B),!.
ev(zero, B) ←!, bdd_zero(B).
ev(one, B) ←!, bdd_one(B).
ev(c(V, N), B) ←!, bdd_equality(V, N, B).
ev(and(A, B), C) ←!, ev(A, BA), ev(B, BB),
    bdd_and(BA, BB, C).
ev(or(A, B), C) ←!, ev(A, BA), ev(B, BB),
    bdd_or(BA, BB, C).
ev(not(A), C) ← ev(A, B), bdd_not(B, C).

```

Figure 5.18 Code for the *ev/2* predicate of PITA(OPT).

are combined in progressively larger terms that are used to check the assumptions, while the probabilities of the combinations are computed only from the probabilities of the operands without considering their structure.

When the program does not satisfy neither assumption, PITA(OPT) can still be beneficial since it delays the construction of BDDs as much as possible and may lead to the construction of less intermediate BDDs than PITA. While in PITA the BDD for each intermediate subgoal must be kept in memory because it is stored in the table and has to be available for future use, in PITA(OPT), BDDs are built only when necessary, leading to a smaller memory footprint and a leaner memory management.

5.9.2 MPE with PITA

MPE inference can be computed by PITA(IND,EXC) by modifying it so that the probability data structure includes the most probable explanation for the subgoal besides the highest probability of the subgoal. In this case, the support predicates are modified as follows:

$$\begin{aligned} & \text{equality}(R, S, Probs, N, e([(R, S, N)], P)) \leftarrow nth(N, Probs, P). \\ & \text{or}(e(E1, P1), e(_E2, P2), e(E1, P1)) \leftarrow P1 \geq P2, !. \\ & \text{or}(e(_E1, _P1), e(E2, P2), e(E2, P2)). \\ & \text{and}(e(E1, P1), e(E2, P2), e(E3, P3)) \leftarrow P3 \text{ is } P1 * P2, \\ & \quad \text{append}(E1, E2, E3). \\ & \text{zero}(e(\text{null}, 0)). \\ & \text{one}(e([], 1)). \\ & \text{ret_prob}(B, B). \end{aligned}$$

In this way, we obtain PITAVIT(IND), so called because for a program encoding an HMM it computes the *Viterbi path*, the sequence of states that most likely originated the output sequence. PITAVIT(IND) is also sound if the exclusiveness assumption does not hold.

5.10 Inference for Queries with an Infinite Number of Explanations

When a discrete program contains function symbols, the number of explanations may be infinite and the probability of the query may be the sum of a convergent series. In this case, the inference algorithm has to recognize the presence of an infinite number of explanations and identify the terms of the series. Sato and Meyer [2012, 2014] extended PRISM by considering programs under the *generative exclusiveness condition*: at any choice point

in any execution path of the top-goal, the choice is done according to a value sampled from a PRISM probabilistic switch. The generative exclusiveness condition implies the exclusive-or condition and that every disjunction originates from a probabilistic choice made by some switch.

In this case, a *cyclic explanation graph* can be computed that encodes the dependency of atoms on probabilistic switches. From this, a system of equations can be obtained defining the probability of ground atoms. Sato and Meyer [2012, 2014] show that first assigning all atoms probability 0 and then repeatedly applying the equations to compute updated values result in a process that converges to a solution of the system of equations. For some program, such as those computing the probability of prefixes of strings from Probabilistic Context-Free Grammars (PCFGs), the system is linear, so solving it is even simpler. In general, this provides an approach for performing inference when the number of explanations is infinite but the generative exclusiveness condition holds.

Gorlin et al. [2012] present the algorithm PIP (for Probabilistic Inference Plus), which is able to perform inference even when explanations are not necessarily mutually exclusive and the number of explanations is infinite. They require the programs to be *temporally well-formed*, i.e., that one of the arguments of predicates can be interpreted as a time that grows from head to body. In this case, the explanations for an atom can be represented succinctly by Definite Clause Grammars (DCGs). Such DCGs are called *explanation generators* and are used to build Factored Explanation Diagrams (FEDs) that have a structure that closely follows that of BDDs. From FEDs, one can obtain a system of polynomial equations that is monotonic and thus convergent as in [Sato and Meyer, 2012, 2014]. So, even when the system is non-linear, a least solution can be computed to within an arbitrary approximation bound by an iterative procedure.

5.11 Inference for Hybrid Programs

Islam et al. [2012b]; Islam [2012] propose an algorithm for performing the DISTR task for Extended PRISM, see Section 4.3. Since it is impossible to enumerate all explanations for the query because there is an uncountable number of them, the idea is to represent derivations symbolically.

Definition 37 (Symbolic derivation [Islam et al., 2012b]). *A goal g directly derives goal g' , denoted $g \rightarrow g'$, if one of the following conditions holds*

PCR if $g = q_1(X_1), g_1$, and there exists a clause in the program, $q_1(Y) \leftarrow r_1(Y_1), r_2(Y_2), \dots, r_m(Y_m)$ such that $\theta = mgu(q_1(X_1), q_1(Y))$ then $g' = (r_1(Y_1), r_2(Y_2), \dots, r_m(Y_m), g_1)\theta$

MSW if $g = msw(rv(X), Y), g_1$ then $g' = g_1$

CONSTR if $g = Constr, g_1$ and *Constr* is satisfiable: then $g' = g_1$.

where **PCR** stands for program clause resolution. A symbolic derivation of g is a sequence of goals g_0, g_1, \dots such that $g = g_0$ and, for all $i \geq 0$, $g_i \rightarrow g_{i+1}$.

Example 73 (Symbolic derivation). Consider Example 56 that we repeat here for ease of reading.

```

widget(X) ←
  msw(m, M), msw(st(M), Z), msw(pt, Y), X = Y + Z.
values(m, [a, b]).
values(st(_), real).
values(pt, real).
← set_sw(m, [0.3, 0.7]).
← set_sw(st(a), norm(2.0, 1.0)).
← set_sw(st(b), norm(3.0, 1.0)).
← set_sw(pt, norm(0.5, 0.1)).

```

The symbolic derivation for goal $widget(X)$ is

$$\begin{array}{c}
 g_1 : widget(X) \\
 \downarrow \\
 g_2 : msw(m, M), msw(st(M), Z), msw(pt, Y), X = Y + Z \\
 \downarrow \\
 g_3 : msw(st(M), Z), msw(pt, Y), X = Y + Z \\
 \downarrow \\
 g_4 : msw(pt, Y), X = Y + Z \\
 \downarrow \\
 g_5 : X = Y + Z \\
 \downarrow \\
 true
 \end{array}$$

Given a goal, the aim of inference is to return a probability density function over the variables of the goal. To do so, all successful symbolic derivations are collected. Then a representation of the probability density associated with the variables of each goal is built bottom-up starting from the leaves. This representation is called a *success function*.

First, we need to identify, for each goal g_i in a symbolic derivation, the set of its *derivation variables* $V(g_i)$, the set of variables appearing as parameters or outcomes of *msws* in some subsequent goal g_j , $j > i$. V is further partitioned into two disjoint sets, V_c and V_d , representing continuous and discrete variables, respectively.

Definition 38 (Derivation variables [Islam et al., 2012b]). *Let $g \rightarrow g'$ such that g' is derived from g using*

PCR *Let θ be the mgu in this step. Then $V_c(g)$ and $V_d(g)$ are the largest set of variables in g such that $V_c(g)\theta \subseteq V_c(g')$ and $V_d(g)\theta \subseteq V_d(g')$*

MSW *Let $g = msw(rv(\mathbf{X}), Y), g_1$. Then $V_c(g)$ and $V_d(g)$ are the largest set of variables in g such that $V_c(g)\theta \subseteq V_c(g') \cup \{Y\}$ and $V_d(g)\theta \subseteq V_d(g') \cup \mathbf{X}$ if Y is continuous, otherwise $V_c(g)\theta \subseteq V_c(g')$ and $V_d(g)\theta \subseteq V_d(g') \cup \mathbf{X} \cup \{Y\}$*

CONSTR *Let $g = Constr, g_1$. Then $V_c(g)$ and $V_d(g)$ are the largest set of variables in g such that $V_c(g)\theta \subseteq V_c(g') \cup vars(Constr)$ and $V_d(g)\theta \subseteq V_d(g')$*

So $V(g)$ is built from $V(g')$ and it can be computed for all goals in a symbolic derivation bottom-up.

Let \mathbf{C} denote the set of all linear equality constraints using set of variables \mathbf{V} and let \mathbf{L} be the set of all linear functions over \mathbf{V} . Let $\mathcal{N}_X(\mu, \sigma^2)$ be the PDF of a univariate Gaussian distribution with mean μ and variance σ^2 , and $\delta_x(X)$ be the Dirac delta function which is zero everywhere except at x and integration of the delta function over its entire range is 1. A *Product Probability Density Function (PPDF)* ϕ over \mathbf{V} is an expression of the form

$$\phi = k \cdot \prod_l \delta_v(V_l) \prod_i \mathcal{N}_{f_i}(\mu_i, \delta_i^2)$$

where k is a non-negative real number, $V_l \in \mathbf{V}$, $f_i \in \mathbf{L}$. A pair (ϕ, C) where $C \subseteq \mathbf{C}$ is called a *constrained PPDF*. A sum of a finite number of constrained PPDFs is called a *success function*, represented as

$$\psi = \sum_i (\phi_i, C_i)$$

We use $C_i(\psi)$ to denote the constraints (i.e., C_i) in the i -th constrained PPDF of success function ψ and $D_i(\psi)$ to denote the i -th PPDF of ψ .

The success function of the query is built bottom-up from the set of derivations for it. The success function of a constraint C is $(1, C)$. The success function of $true$ is $(1, true)$. The success function of $msw(rv(\mathbf{X}), Y)$ is $(\psi, true)$ where ψ is the probability density function of rv 's distribution if rv is continuous, and its probability mass function if rv is discrete.

Example 74 (Success functions of msw atoms). *The success function of $msw(m, M)$ for the program in Example 73 is*

$$\psi_{msw(m,M)}(M) = 0.3\delta_a(M) + 0.7\delta_b(M)$$

We can represent success functions using tables, where each table row denotes discrete random variable valuations. For example, the above success function can be represented as

M	$\psi_{msw(m,M)}(M)$
a	0.3
b	0.7

For a $g \rightarrow g'$ derivation step, the success function of g is computed from the success function of g' using the join and marginalization operations, the first for **MSW** and **CONSTR** steps and the latter for **PCR** steps.

Definition 39 (Join operation). *Let $\psi_1 = \sum_i(D_i, C_i)$ and $\psi_2 = \sum_j(D_j, C_j)$ be two success functions, then the join $\psi_1 * \psi_2$ of ψ_1 and ψ_2 is the success function*

$$\sum_{i,j}(D_i D_j, C_i \wedge C_j)$$

Example 75 (Join operation). *Let $\psi_{msw(m,M)}(M)$ and $\psi_g(X, Y, Z, M)$ be defined as follows:*

M	$\psi_{msw(m,M)}(M)$	M	$\psi_g(X, Y, Z, M)$
a	0.3	a	$(\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$
b	0.7	b	$(\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$

The join of $\psi_{msw(m,M)}(M)$ and $\psi_g(X, Y, Z, M)$ is:

M	$\psi_{msw(m,M)}(M) * \psi_g(X, Y, Z, M)$
a	$(0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$
b	$(0.7\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$

Since $\delta_a(M)\delta_b(M) = 0$ because M cannot be both a and b at the same time, we simplified the result by eliminating any such inconsistent PPDF term in ψ .

In the case of a **PCR** derivation step $g \rightarrow g'$, g may contain a subset of the variables of g' . To compute the success function for g , we thus must marginalize over the eliminated variables. If an eliminated variable is discrete, marginalization is done by summation in the obvious way. If an eliminated variable V is continuous, marginalization is done in two steps: projection and integration. The goal of projection is to eliminate any linear constraint on V . The projection operation finds a linear constraint $V = \mathbf{a}\mathbf{X} + b$ on V and replaces all occurrences of V in the success function by $\mathbf{a}\mathbf{X} + b$.

Definition 40 (Projection of a success function). *The projection of a success function ψ w.r.t. a continuous variable V , denoted by $\psi \downarrow_V$, is a success function ψ' such that $\forall i$:*

$$D_i(\psi') = D_i(\psi)[V/\mathbf{a}\mathbf{X} + b]$$

and

$$C_i(\psi') = (C_i(\psi) - C_{ip})[V/\mathbf{a}\mathbf{X} + b]$$

where C_{ip} is a linear constraint $V = \mathbf{a}\mathbf{X} + b$ on V in $C_i(\psi)$ and $t[x/s]$ denotes the replacement of all occurrences of x in t by s .

If ψ does not contain any linear constraint on V , then the projected form remains the same.

Example 76 (Projection operation). *Let ψ_1 be the success function*

$$\psi_1 = (0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$$

The projection of ψ_1 w.r.t. Y is

$$\psi_1 \downarrow_Y = 0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_{X-Z}(0.5, 0.1), true)$$

We can now define the integration operation.

Definition 41 (Integration of a success function). *Let ψ be a success function that does not contain any linear constraints on V . Then the integration of ψ w.r.t. to V , denoted by $\int_V \psi$, is the success function ψ' such that*

$$\forall i : D_i(\psi') = \int D_i(\psi)dV$$

Islam et al. [2012b]; Islam [2012] prove that the integration of a PPDF with respect to a variable V is a PPDF, i.e., that

$$\alpha \int_{-\infty}^{+\infty} \prod_{k=1}^m \mathcal{N}_{\mathbf{a}_k \mathbf{X}_k + b_k}(\mu_k, \sigma_k^2) dV = \alpha' \prod_{l=1}^{m'} \mathcal{N}_{\mathbf{a}'_l \mathbf{X}'_l + b'_l}(\mu'_l, \sigma'^2_l)$$

where $V \in \mathbf{X}_k$ and $V \notin \mathbf{X}'_l$.

For example, the integration of $\mathcal{N}_{a_1 V - X_1}(\mu_1, \sigma_1^2) \mathcal{N}_{a_2 V - X_2}(\mu_2, \sigma_2^2)$ w.r.t. variable V is

$$\int_{-\infty}^{+\infty} \mathcal{N}_{a_1 V - X_1}(\mu_1, \sigma_1^2) \mathcal{N}_{a_2 V - X_2}(\mu_2, \sigma_2^2) dV = \mathcal{N}_{a_2 X_1 - a_1 X_2}(a_1 \mu_1 - a_2 \mu_1, a_2^2 \sigma_1^2 + a_1^2 \sigma_2^2) \tag{5.3}$$

where X_1 and X_2 are linear combinations of variables except V .

Example 77 (Integration of a success function). *Let ψ_2 represent the following success function*

$$\psi_2 = (0.3 \mathcal{N}_Z(2.0, 1.0) \mathcal{N}_{X-Z}(0.5, 0.1), true)$$

Then integration of ψ_2 w.r.t. Z yields

$$\oint_Z \psi_2 = \left(\int 0.3 \mathcal{N}_Z(2.0, 1.0) \mathcal{N}_{X-Z}(0.5, 0.1), true \right) = (0.3 \mathcal{N}_X(2.5, 1.1), true)$$

by Equation (5.3).

The marginalization operation is the composition of the join and integration operations.

Definition 42 (Marginalization of a success function). *The marginalization of a success function ψ with respect to a variable V , denoted by $\mathbb{M}(\psi, V)$, is a success function ψ' such that*

$$\psi' = \oint_V \psi \downarrow_V$$

The marginalization over a set of variables is defined as $\mathbb{M}(\psi, \{V\} \cup \mathbf{X}) = \mathbb{M}(\mathbb{M}(\psi, V), \mathbf{X})$ and $\mathbb{M}(\psi, \emptyset) = \psi$.

The set of all success functions is closed under join and marginalization operations. The success function for a derivation can now be defined as follows.

Definition 43 (Success function of a goal). *The success function of a goal g , denoted by ψ_g , is computed based on the derivation $g \rightarrow g'$:*

$$\psi_g = \begin{cases} \sum_{g'} \mathbb{M}(\psi_{g'}, V(g') - V(g)) & \text{for all PCR } g \rightarrow g' \\ \psi_{msw(rv(\mathbf{X}), Y)} * \psi_{g'} & \text{if } g = msw(rv(\mathbf{X}), Y), g_1 \\ \psi_{Constr} * \psi_{g'} & \text{if } g = Constr, g_1 \end{cases}$$

Example 78 (Success function of a goal). *Consider the symbolic derivation of Example 73. The success function of goal g_5 is $\psi_{g_5}(X, Y, Z) = (1, X = Y + Z)$. To obtain ψ_{g_4} , we must perform a join operation:*

$$\psi_{g_4}(X, Y, Z) = \psi_{msw(pt, Y)}(Y) * \psi_{g_5}(X, Y, Z) = (\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$$

*The success function of goal g_3 is $\psi_{msw(st(M), Z)}(Z) * \psi_{g_4}(X, Y, Z)$:*

M	$\psi_{g_3}(X, Y, Z, M)$
a	$(\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$
b	$(\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$

Then we join $\psi_{msw(m, M)}(M)$ and $\psi_{g_3}(X, Y, Z, M)$:

M	$\psi_{g_2}(X, Y, Z, M)$
a	$(0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$
b	$(0.7\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z)$

The success function of g_1 is $\psi_{g_1}(X) = \mathbb{M}(\psi_{g_2}(X, Y, Z, M), \{M, Y, Z\})$. We marginalize $\psi_{g_2}(X, Y, Z, M)$ first w.r.t. M :

$$\begin{aligned} \psi'_{g_2} &= \mathbb{M}(\psi_{g_2}, M) = \oint_M \psi_{g_2} \downarrow_M = \\ &= (0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z) + \\ &\quad (0.7\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_Y(0.5, 0.1), X = Y + Z) \end{aligned}$$

Then we marginalize $\psi'_{g_2}(X, Y, Z)$ w.r.t. Y :

$$\begin{aligned} \psi''_{g_2} &= \mathbb{M}(\psi'_{g_2}, Y) = \oint_Y \psi'_{g_2} \downarrow_Y = \\ &= 0.3\mathcal{N}_Z(2.0, 1.0)\mathcal{N}_{X-Z}(0.5, 0.1) + \\ &\quad 0.7\mathcal{N}_Z(3.0, 1.0)\mathcal{N}_{X-Z}(0.5, 0.1) \end{aligned}$$

Finally, we get $\psi_{g_1}(X)$ by marginalizing $\psi''_{g_2}(X, Z)$ w.r.t. Z :

$$\begin{aligned}\psi_{g_1}(X) &= \mathbb{M}(\psi''_{g_2}, Z) = \int_Z \psi''_{g_2} \downarrow_Z = \\ &= 0.3\mathcal{N}_Z(2.5, 1.1) + 0.7\mathcal{N}_X(3.5, 1.1)\end{aligned}$$

So the algorithm returns the probability density of the continuous random variables in the query.

This algorithm is correct if the program satisfies PRISM's assumptions: independent-and and exclusive-or. The first is equivalent to requiring that an instance of a random variable occurs at most once in any derivation. In fact, the join operation used in a $g \rightarrow g'$ **MSW** step is correct only if the random variable defined by the *msw* atom does not appear in g' . Moreover, the sum over all **PCR** steps in Definition 43 is correct only if the terms are mutually exclusive.

A subgoal may appear more than once in the derivation tree for a goal, so tabling can be effectively used to avoid redundant computation.

6

Lifted Inference

Reasoning with real-world models is often very expensive due to their complexity. However, sometimes the cost can be reduced by exploiting symmetries in the model. This is the task of *lifted* inference, that answers queries by reasoning on populations of individuals as a whole instead of considering each entity individually. The exploitation of the symmetries in the model can significantly speed up inference.

Lifted inference was initially proposed by Poole [2003]. Since then, many techniques have appeared such as lifted versions of variable elimination and belief propagation, using approximations and dealing with models such as parfactor graphs and MLNs [de Salvo Braz et al., 2005; Milch et al., 2008; Van den Broeck et al., 2011].

6.1 Preliminaries on Lifted Inference

Applying lifted inference to PLP languages under the DS is problematic because the conclusions of different rules are combined with noisy-OR that requires aggregations at the lifted level when existential variables are present. For example, consider the following ProbLog program from [De Raedt and Kimmig, 2015]:

```
p :: famous(Y) .
popular(X) :- friends(X, Y), famous(Y) .
```

In this case, $P(\text{popular}(\text{john})) = 1 - (1 - p)^m$ where m is the number of friends of `john`. This is because the body contains a logical variable not appearing in the head, which is thus existentially quantified. A grounding of the atom in the head of this clause represents the disjunction of a number of ground bodies. In this case, we don't need to know the identities of these friends, we just need to know how many there are. Hence, we don't need to ground the clauses.

Example 79 (Running example for lifted inference – ProbLog). *We consider a ProbLog program representing the workshop attributes problem of [Milch et al., 2008]. It models the organization of a workshop where a number of people have been invited. The predicate `series` indicates whether the workshop is successful enough to start a series of related meetings while `attends(P)` indicates whether person `P` will attend the workshop. We can model this problem with the ProbLog program:*

```
series :- self.
series :- attends(P) .
attends(P) :- at(P,A) .
0.1::self.
0.3::at(P,A) :- person(P), attribute(A) .
```

Note that all rules are range-restricted, i.e., all variables in the head also appear in a positive literal in the body. A workshop becomes a series either because of its own merits with a 10% probability (represented by the probabilistic fact `self`) or because people attend. People attend the workshop depending on the workshop’s attributes such as location, date, fame of the organizers, etc. (modeled by the probabilistic fact `at(P,A)`). The probabilistic fact `at(P,A)` represents whether person `P` attends because of attribute `A`. Note that the last statement corresponds to a set of ground probabilistic facts, one for each person `P` and attribute `A` as in ProbLog2 (Section 5.7). For the sake of brevity, we omit the (non-probabilistic) facts describing the `person/1` and `attribute/1` predicates.

Parameterized Random Variables (PRVs) and *parfactors* have been defined in Section 2.10.3. We briefly recall here their definition. PRVs represent sets of random variables, one for each possible ground substitution to all of its logical variables. Parfactors are triples

$$\langle \mathcal{C}, \mathcal{V}, F \rangle$$

where \mathcal{C} is a set of inequality constraints on logical variables, \mathcal{V} is a set of PRVs and F is a factor that is a function from the Cartesian product of ranges of PRVs of \mathcal{V} to real values. A parfactor is also represented as $F(\mathcal{V})|\mathcal{C}$ or $F(\mathcal{V})$ if there are no constraints. A *constrained PRV* V is of the form $V|\mathcal{C}$, where $V = P(X_1, \dots, X_n)$ is a non-ground atom and \mathcal{C} is a set of constraints on logical variables $\mathbf{X} = \{X_1, \dots, X_n\}$. Each constrained PRV represents the set of random variables $\{P(\mathbf{x})|\mathbf{x} \in \mathcal{C}\}$, where \mathbf{x} is the tuple of constants (x_1, \dots, x_n) . Given a (constrained) PRV V , we use $RV(V)$ to denote the set

of random variables it represents. Each ground atom is associated with one random variable, which can take any value in $range(V)$.

PFL [Gomes and Costa, 2012] described in Section 2.10.3 extends Prolog to support probabilistic reasoning with parametric factors. We repeat below Example 36 for ease of reading.

Example 80 (Running example – PFL program). *A version of the workshop attributes problem can be modeled by a PFL program such as*

```
bayes series, attends(P); [0.51, 0.49, 0.49, 0.51];
    [person(P)].
bayes attends(P), at(P,A); [0.7, 0.3, 0.3, 0.7];
    [person(P), attribute(A)].
```

The first PFL factor has series and attends(P) as Boolean random variable arguments, [0.51, 0.49, 0.49, 0.51] as table and [person(P)] as constraint.

This model is not equivalent to the one of Example 79, but it corresponds to a ProbLog program that has only the second and the third clause of Example 79. Models equivalent to Example 79 will be given in Examples 82 and 83.

6.1.1 Variable Elimination

Variable Elimination (VE) [Zhang and Poole, 1994, 1996] is an algorithm for probabilistic inference on graphical models. VE takes as input a set of factors \mathcal{F} , an elimination order ρ , a query variable X , and a list \mathbf{y} of observed values. After setting the observed variables in all factors to their corresponding observed values, VE eliminates the random variables from the factors one by one until only the query variable X remains. This is done by selecting the first variable Z from the elimination order ρ and then calling SUM-OUT that eliminates Z by first multiplying all the factors that include Z into a single factor and summing out Z from the newly constructed factor. This procedure is repeated until ρ becomes empty. In the final step, VE multiplies together the factors of \mathcal{F} obtaining a new factor γ that is normalized as $\gamma(x)/\sum_{x'}\gamma(x')$ to give the posterior probability.

In many cases, we need to represent factors where a Boolean variable X with parents \mathbf{Y} is true if any of the Y_i is true, i.e., the case where X is the disjunction of the variables in \mathbf{Y} . This may, for example, represent the situation where the Y_i s are causes of X , each capable of making X true

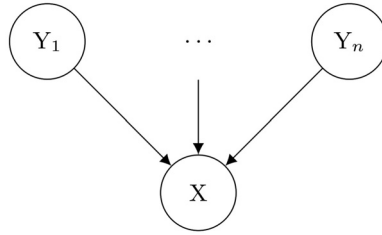


Figure 6.1 BN representing an OR dependency between X and Y.

independently of the values of the others. This is represented by the BN of Figure 6.1 where the CPT for X is deterministic and is given by

	At least one Y_i equal to 1	Remaining columns
$X = 1$	1.0	0.0
$X = 0$	0.0	1.0

In practice, however, each parent Y_i may have a noisy inhibitor variable I_i that independently blocks or activates Y_i , so X is true if either *any* of the causes Y_i holds true *and* is not inhibited. This can be represented with the BN of Figure 6.2 where the Y'_i are given by the Boolean formula $Y'_i = Y_i \wedge \neg I_i$, i.e., Y'_i is true if Y_i is true and is not inhibited. So the CPT for the Y'_i is deterministic. The I_i variables have no parents and their CPT is $P(I_i = 0) = \Pi_i$, where Π_i is the probability that Y_i is not inhibited.

This represents the fact that X is not simply the disjunction of Y but depends probabilistically from Y with a factor that encodes the probability that the Y_i variables are inhibited.

If we marginalize over the I_i variables, we obtain a BN like the one of Figure 6.1 where, however, the CPT for X is not anymore that of a disjunction but takes into account the probabilities that the parents are inhibited. This is called a *noisy-OR gate*. Handling this kind of factor is a non-trivial problem. Noisy-OR gates are also called *causal independent* models. An example of an entry in a noisy-OR factor is

$$P(X = 1 | Y_1 = 1, \dots, Y_n = 1) = 1 - \prod_{i=1}^n (1 - \Pi_i)$$

In fact, X is true iff none of its causes is inhibited.

The factor for a noisy-OR can be expressed as a combination of factors by using the intermediate Y'_i variables that represent the effect of each cause taking into account the inhibitor.

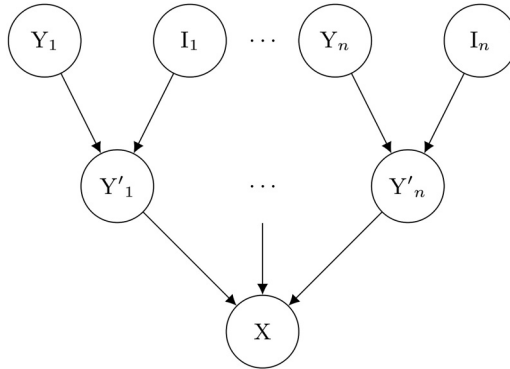


Figure 6.2 BN representing a noisy-OR dependency between X and \mathbf{Y} .

For example, suppose X has two causes Y_1 and Y_2 and let $\phi(y_1, y_2, x)$ be the noisy-OR factor. Let the variables Y'_1 and Y'_2 be defined as in Figure 6.2 and consider the factors $\psi_1(y_1, y'_1)$ and $\psi_2(y_2, y'_2)$ modeling the dependency of Y'_i from Y_i . These factors are obtained by marginalizing the I_i variables, so, if $P(I_i = 0) = \Pi_i$, they are given by

$\psi(y_1, y'_1)$	$Y_i = 1$	$Y_i = 0$
$Y'_i = 1$	Π_i	0.0
$Y'_i = 0$	$1 - \Pi_i$	1.0

Then the factor $\phi(y_1, y_2, x)$ can be expressed as

$$\phi(y_1, y_2, x) = \sum_{y'_1 \vee y'_2 = x} \psi_1(y_1, y'_1) \psi_2(y_2, y'_2) \tag{6.1}$$

where the summation is over all values y'_1 and y'_2 of Y'_1 and Y'_2 whose disjunction is equal to x . The X variable is called *convergent* as it is where independent contributions from different sources are collected and combined. Non-convergent variables will be called *regular variables*.

Representing factors such as ϕ with ψ_1 and ψ_2 is advantageous when the number of parents grows large, as the combined size of the component factors grows linearly, instead of exponentially.

Unfortunately, a straightforward use of VE for inference would lead to construct $O(2^n)$ tables where n is the number of parents and the summation in Equation (6.1) will have an exponential number of terms. A modified algorithm, VE1 [Zhang and Poole, 1996], combines factors through a new operator \otimes :

$$\phi \otimes \psi(E_1 = \alpha_1, \dots, E_k = \alpha_k, \mathbf{A}, \mathbf{B}_1, \mathbf{B}_2) = \sum_{\alpha_{11} \vee \alpha_{12} = \alpha_1} \dots \sum_{\alpha_{k1} \vee \alpha_{k2} = \alpha_k} \phi(E_1 = \alpha_{11}, \dots, E_k = \alpha_{k1}, \mathbf{A}, \mathbf{B}_1) \psi(E_1 = \alpha_{12}, \dots, E_k = \alpha_{k2}, \mathbf{A}, \mathbf{B}_2) \quad (6.2)$$

Here, ϕ and ψ are two factors that share convergent variables $E_1 \dots E_k$, \mathbf{A} is the list of regular variables that appear in both ϕ and ψ , while \mathbf{B}_1 and \mathbf{B}_2 are the lists of variables appearing only in ϕ and ψ , respectively. By using the \otimes operator, factors encoding the effect of parents can be combined in pairs, without the need to apply Equation (6.1) on all factors at once.

Factors containing convergent variables are called *heterogeneous* while the remaining factors are called *homogeneous*. Heterogeneous factors sharing convergent variables must be combined with the operator \otimes , called *heterogeneous multiplication*.

Algorithm VE1 exploits causal independence by keeping two lists of factors: a list of homogeneous factors \mathcal{F}_1 and a list of heterogeneous factors \mathcal{F}_2 . Procedure SUM-OUT is replaced by SUM-OUT1 that takes as input \mathcal{F}_1 and \mathcal{F}_2 and a variable Z to be eliminated. First, all the factors containing Z are removed from \mathcal{F}_1 and combined with multiplication to obtain factor ϕ . Then all the factors containing Z are removed from \mathcal{F}_2 and combined with heterogeneous multiplication obtaining ψ . If there are no such factors $\psi = nil$. In the latter case, SUM-OUT1 adds the new (homogeneous) factor $\sum_z \phi$ to \mathcal{F}_1 ; otherwise, it adds the new (heterogeneous) factor $\sum_z \phi\psi$ to \mathcal{F}_2 . Procedure VE1 is the same as VE with SUM-OUT replaced by SUM-OUT1 and with the difference that two sets of factors are maintained instead of one.

However, VE1 is not correct for any elimination order. Correctness can be ensured by *deputizing* the convergent variables: every such variable X is replaced by a new convergent variable X' (called a *deputy variable*) in the heterogeneous factors containing it, so that X becomes a regular variable. Finally, a new factor $\iota(X, X')$ is introduced, called *deputy factor*, that represents the identity function between X and X' , i.e., it is defined by

$\iota(X, X')$	00	01	10	11
	1.0	0.0	0.0	1.0

The network on which VE1 works thus takes the form shown in Figure 6.3. Deputizing ensures that VE1 is correct as long as the elimination order is such that $\rho(X') < \rho(X)$.

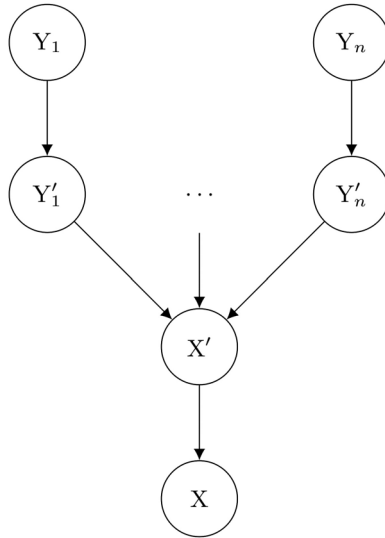


Figure 6.3 BN of Figure 6.1 after deputation.

6.1.2 GC-FOVE

Work on lifting VE started with FOVE [Poole, 2003] and led to the definition of C-FOVE [Milch et al., 2008]. C-FOVE was refined in GC-FOVE [Taghipour et al., 2013], which represents the state of the art. Then, Gomes and Costa [Gomes and Costa, 2012] adapted GC-FOVE to PFL.

First-order variable elimination (FOVE) [Poole, 2003; de Salvo Braz et al., 2005] computes the marginal probability distribution for a query random variable by repeatedly applying operators that are lifted counterparts of VE’s operators. Models are in the form of a set of parfactors that are essentially the same as in PFL.

GC-FOVE tries to eliminate all (non-query) PRVs in a particular order by applying the following operations:

1. *Lifted Sum-Out* that excludes a PRV from a parfactor ϕ if the PRVs only occurs in ϕ ;
2. *Lifted Multiplication* that multiplies two aligned parfactors. Matching variables must be properly aligned and the new coefficients must be computed taking into account the number of groundings in the constraints \mathcal{C} ;
3. *Lifted Absorption* that eliminates n PRVs that have the same observed value.

If these operations cannot be applied, an enabling operation must be chosen such as *splitting* a parfactor so that some of its PRVs match another parfactor. If no operation can be executed, GC-FOVE completely grounds the PRVs and parfactors and performs inference on the ground level.

GC-FOVE also considers PRVs with counting formulas, introduced in C-FOVE [Milch et al., 2008]. A counting formula takes advantage of symmetries existing in factors that are products of independent variables. It allows the representation of a factor of the form $\phi(P(x_1), P(x_2), \dots, P(x_n))$, where all PRVs have the same domain, as $\phi(\#_X[P(X)])$, where $\#_X[P(X)]$ is the *counting formula*. The factor implements a multinomial distribution, such that its values depend on the number of variables n and the domain size. Counting formulas may result from summing-out, when we obtain parfactors with a single PRV, or through *Counting Conversion* that searches for factors of the form

$$\phi\left(\prod_i (S(X_j)P(X_j, Y_i))\right)$$

and counts on the occurrences of Y_i .

GC-FOVE employs a constraint-tree to represent arbitrary constraints \mathcal{C} , whereas PFL simply uses sets of tuples. Arbitrary constraints can capture more symmetries in the data, which potentially offers the ability to perform more operations at a lifted level.

6.2 LP²

LP² [Bellodi et al., 2014] is an algorithm for performing lifted inference in ProbLog that translates the program into PFL and uses an extended GC-FOVE version for managing noisy-OR nodes.

6.2.1 Translating ProbLog into PFL

In order to translate ProbLog into PFL, the program must be acyclic (Definition 4, see Section 6.5 for the case of cyclic programs). If this condition is fulfilled, the ProbLog program can be converted first into a BN with noisy-OR nodes. Here we specialize the conversion for LPADs presented in Section 2.5 to the case of ProbLog.

For each atom A in the Herbrand base of the program, the BN contains a Boolean random variable with the same name. Each probabilistic fact $p :: a$ is represented by a parentless node with the CPT:

a	0	1
	$1 - p$	p

For each ground rule $R_i = h \leftarrow b_1, \dots, b_n, \sim c_1, \dots, \sim c_m$, we add to the network a random variable called H_i that has as parents the random variables representing the atoms $B_1, \dots, B_n, C_1, \dots, C_m$ and the following CPT:

H_i	$B_1 = 1, \dots, B_n = 1, C_1 = 0, \dots, C_m = 0$	all other columns
0	0.0	1.0
1	1.0	0.0

In practice, H_i is the result of the conjunction of the random variables representing the atoms in the body. Then, for each ground atom h in the Herbrand base not appearing in a probabilistic fact, we add random variable H to the network, with all the H_i s of the ground rules with H in the head as parents and with CPT:

H	At least one $H_i = 1$	All other columns
f	0.0	1.0
t	1.0	0.0

representing the result of the disjunction of the random variables H_i . These families of random variables can be directly represented in PFL without the need to first ground the program, thus staying at the lifted level.

Example 81 (Translation of a ProbLog program into PFL). *The translation of the ProbLog program of Example 79 into PFL is*

```

bayes series1, self; [1, 0, 0, 1] ; [].
bayes series2, attends(P); [1, 0, 0, 1];
    [person(P)].
bayes series, series1, series2 ; [1, 0, 0, 0, 0,
    1, 1, 1]; [].
bayes attends1(P), at(P,A); [1, 0, 0, 1];
    [person(P), attribute(A)].
bayes attends(P), attends1(P); [1, 0, 0, 1];
    [person(P)].
bayes self; [0.9, 0.1]; [].
bayes at(P,A); [0.7, 0.3] ; [person(P),
    attribute(A)].

```


Notice that `series2` and `attends1(P)` can be seen as or-nodes, since they are in fact convergent variables. Thus, after grounding, factors derived from the second and the fourth parfactor should not be multiplied together but should be combined with heterogeneous multiplication.

To do so, we need to identify heterogeneous factors and add deputy variables and parfactors. We thus introduce two new types of parfactors to PFL, `het` and `deputy`. As mentioned before, the type of a parfactor refers to the type of the network over which that parfactor is defined. These two new types are used in order to define a noisy-OR (Bayesian) network. The first parfactor is such that its ground instantiations are heterogeneous factors. The convergent variables are assumed to be represented by the first atom in the parfactor list of atoms. Lifting identity is straightforward: it corresponds to two atoms with an identity factor between their ground instantiations. Since the factor is fixed, it is not indicated.

Example 82 (ProLog program to PFL – LP²). *The translation of the Prolog program of Example 79, shown in Example 81, is modified with the two new factors `het` and `deputy` as shown below:*

```

bayes series1p, self; [1, 0, 0, 1] ; [].
het series2p, attends(P); [1, 0, 0, 1];
    [person(P)].
deputy series2, series2p; [].
deputy series1, series1p; [].
bayes series, series1, series2; [1, 0, 0, 0, 0, 1,
    1, 1] ; [].
het attends1p(P), at(P,A); [1, 0, 0, 1];
    [person(P), attribute(A)].
deputy attends1(P), attends1p(P); [person(P)].
bayes attends(P), attends1(P); [1, 0, 0, 1];
    [person(P)].
bayes self; [0.9, 0.1]; [].
bayes at(P,A); [0.7, 0.3] ; [person(P),
    attribute(A)].

```

Here, `series1p`, `series2p`, and `attends1p(P)` are the new convergent deputy random variables, and `series1`, `series2`, and `attends1(P)` are their corresponding regular variables. The fifth factor represents the OR combination of `series1` and `series2` to variable `series`.

GC-FOVE must be modified in order to take into account heterogeneous parfactors and convergent PRVs. The VE algorithm must be replaced by VE1, i.e., two lists of factors must be maintained, one with homogeneous and the other with heterogeneous factors. When eliminating variables, homogeneous factors have higher priority and are combined with homogeneous factors only. Then heterogeneous factors are taken into account and combined before starting to mix factors from both types, to produce a final factor from which the selected random variable is eliminated.

Lifted heterogeneous multiplication considers the case in which the two factors share convergent random variables. The SUM-OUT operator must be modified as well to take into account the case that random variables must be summed out from a heterogeneous factor. The formal definition of these two operators is rather technical and we refer to [Bellodi et al., 2014] for the details.

6.3 Lifted Inference with Aggregation Parfactors

Kisynski and Poole [Kisynski and Poole, 2009a] proposed an approach based on *aggregation parfactors* instead of parfactors. Aggregation parfactors are very expressive and can represent different kinds of causal independence models, where noisy-OR and noisy-MAX are special cases. They are of the form $\langle C, P, C, F_P, \boxtimes, C_A \rangle$, where P and C are PRVs which share all the parameters except one – let’s say A which is in P but not in C – and the range of P (possibly non-Boolean) is a subset of that of C ; C and C_A are sets of inequality constraints respectively not involving and involving A ; F_P is a factor from the range of P to real values; and \boxtimes is a commutative and associative deterministic binary operator over the range of C .

When \boxtimes is the MAX operator, of which the OR operator is a special case, a total ordering $<$ on the range of C can be defined. An aggregation parfactor can be replaced with two parfactors of the form $\langle C \cup C_A, \{P, C'\}, F_C \rangle$ and $\langle C, \{C, C'\}, F_\Delta \rangle$, where C' is an auxiliary PRV that has the same parameterization and range as C . Let \mathbf{v} be an assignment of values to random variables, then $F_C(\mathbf{v}(P), \mathbf{v}(C')) = F_P(\mathbf{v}(P))$ when $\mathbf{v}(P) \leq \mathbf{v}(C')$, $F_C(\mathbf{v}(P), \mathbf{v}(C')) = 0$ otherwise, while $F_\Delta(\mathbf{v}(C), \mathbf{v}(C')) = 1$ if $\mathbf{v}(C) = \mathbf{v}(C')$, -1 if $\mathbf{v}(C)$ is equal to a successor of $\mathbf{v}(C')$ and 0 otherwise.

In ProbLog, we can use aggregation parfactors to model the dependency between the head of a rule and the body, when the body contains a single literal with an extra variable. In this case in fact, given a grounding of the head, the contribution of all the ground clauses with that head must be

combined by means of an OR. Since aggregation parfactors are replaced by regular parfactors, the technique can be used to reason with ProbLog by converting the program into PFL with these additional parfactors. The conversion is possible only if the ProbLog program is acyclic.

In the case of ProbLog, the range of PRVs is binary and \boxtimes is OR. For example, the clause `series2:- attends(P)` can be represented with the aggregation parfactor

$$\langle \emptyset, \text{attends}(P), \text{series2}, F_P, \vee, \emptyset \rangle,$$

where $F_P(0) = 1$ and $F_P(1) = 1$. This is replaced by the parfactors

$$\langle \emptyset, \{\text{attends}(P), \text{series2p}\}, F_C \rangle$$

$$\langle \emptyset, \{\text{series2}, \text{series2p}\}, F_\Delta \rangle$$

with $F_C(0,0) = 1$, $F_C(0,1) = 1$, $F_C(1,0) = 0$, $F_C(1,1) = 1$, $F_\Delta(0,0) = 1$, $F_\Delta(0,1) = 0$, $F_\Delta(1,0) = -1$, and $F_\Delta(1,1) = 1$.

When the body of a rule contains more than one literal and/or more than one extra variable with respect to the head, the rule must be first split into multiple rules (adding auxiliary predicate names) satisfying the constraint.

Example 83 (ProbLog program to PFL – aggregation parfactors). *The program of Example 79 using the above encoding for aggregation parfactors is*

```

bayes series1p, self; [1, 0, 0, 1] ; [].
bayes series2p, attends(P) [1, 0, 1, 1];
    [person(P)].
bayes series2, series2p; [1, 0, -1, 1]; [].
bayes series1, series1p; [1, 0, -1, 1]; [].
bayes series, series1, series2; [1, 0, 0, 0, 0, 1,
    1, 1] ; [].
bayes attends1p(P), at(P,A); [1, 0, 1, 1];
    [person(P), attribute(A)].
bayes attends1(P), attends1p(P); [1, 0, -1, 1];
    [person(P)].
bayes attends(P), attends1(P); [1, 0, 0, 1];
    [person(P)].
bayes self; [0.9, 0.1]; [].
bayes at(P,A); [0.7, 0.3] ;
    [person(P), attribute(A)].

```

Thus, by using the technique of [Kisynski and Poole, 2009a], we can perform lifted inference in ProbLog by a simple conversion to PFL, without the need to modify PFL algorithms.

6.4 Weighted First-Order Model Counting

A different approach to lifted inference for PLP uses Weighted First Order Model Counting (WFOMC). WFOMC takes as input a triple (Δ, w, \bar{w}) , where Δ is a sentence in first-order logic and w and \bar{w} are weight functions which associate a real number to positive and negative literals, respectively, depending on their predicate. Given a triple (Δ, w, \bar{w}) and a query ϕ , its probability $P(\phi)$ is given by

$$P(\phi) = \frac{WFOMC(\Delta \wedge \phi, w, \bar{w})}{WFOMC(\Delta, w, \bar{w})}$$

Here, $WFOMC(\Delta, w, \bar{w})$ corresponds to the sum of the weights of all Herbrand models of Δ , where the weight of a model is the product of its literal weights. Hence

$$WFOMC(\Delta, w, \bar{w}) = \sum_{\omega \models \Delta} \prod_{l \in \omega_0} \bar{w}(pred(l)) \prod_{l \in \omega_1} w(pred(l))$$

where ω_0 and ω_1 are, respectively, false and true literals in the interpretation ω and $pred$ maps literals l to their predicate. Two lifted algorithms exist for exact WFOMC, one based on first-order knowledge compilation [Van den Broeck et al., 2011; Van den Broeck, 2011; Van den Broeck, 2013] and the other based on first-order DPLL search [Gogate and Domingos, 2011]. They both require the input theory to be in *first-order CNF*. A first-order CNF is a theory consisting of a conjunction of sentences of the form

$$\forall X_1 \dots, \forall X_n l_1 \vee \dots \vee l_m.$$

A ProbLog program can be encoded as a first-order CNF using Clark's completion, see Section 1.4.1. For acyclic logic programs, Clark's completion is correct, in the sense that every model of the logic program is a model of the completion, and vice versa. The result is a set of rules in which each predicate is encoded by a single sentence. Consider ProbLog rules of the form $p(\mathbf{X}) \leftarrow B_i(\mathbf{X}, Y_i)$ where Y_i is a variable that appears in the body B_i but not in the head $P(\mathbf{X})$. The corresponding sentence in the completion is $\forall \mathbf{X} p(\mathbf{X}) \leftrightarrow \bigvee_i \exists Y_i B_i(\mathbf{X}, Y_i)$. For cyclic programs, see Section 6.5 below.

Since WFOMC requires an input where existential quantifiers are absent, Van den Broeck et al. [2014] presented a sound and modular Skolemization procedure to translate ProbLog programs into first-order CNF. Regular Skolemization cannot be used because it introduces function symbols, that are problematic for model counters. Therefore, existential quantifiers in expressions of the form $\exists X \phi(X, \mathbf{Y})$ are replaced by the following formulas [Van den Broeck et al., 2014]:

$$\forall \mathbf{Y} \forall X z(\mathbf{Y}) \vee \neg \phi(X, \mathbf{Y})$$

$$\forall \mathbf{Y} s(\mathbf{Y}) \vee z(\mathbf{Y})$$

$$\forall \mathbf{Y} \forall X s(\mathbf{Y}) \vee \neg \phi(X, \mathbf{Y})$$

Here z is the Tseitin predicate ($w(z) = \bar{w}(z) = 1$) and s is the Skolem predicate ($w(s) = 1, \bar{w}(s) = -1$). This substitution can also be used for eliminating universal quantifiers since

$$\forall X \phi(X, \mathbf{Y})$$

can be seen as

$$\neg \exists X \neg \phi(X, \mathbf{Y}).$$

Existential quantifiers are removed until no more substitutions can be applied. The resulting program can then be encoded as a first-order CNF with standard transformations.

This replacement introduces a relaxation of the theory, thus the theory admits more models besides the regular, wanted ones. However, for every additional, unwanted model with weight W , there is exactly one additional model with weight $-W$, and thus the WFOMC does not change. The interaction between the three relaxed formulas and the model weights follows the behavior:

1. When $z(\mathbf{Y})$ is false, then $\exists X \phi(X, \mathbf{Y})$ is false while $s(\mathbf{Y})$ is true, this is a regular model whose weight is multiplied by 1.
2. When $z(\mathbf{Y})$ is true, then either:
 - (a) $\exists X \phi(X, \mathbf{Y})$ is true and $s(\mathbf{Y})$ is true, this is a regular model whose weight is multiplied by 1; or
 - (b) $\exists X \phi(X, \mathbf{Y})$ is false and $s(\mathbf{Y})$ is true, this is an additional model with a positive weight W , or
 - (c) $\exists X \phi(X, \mathbf{Y})$ is true and $s(\mathbf{Y})$ is false, this is an additional model with weight $-W$.

The last two cases cancel out.

The WFOMC encoding for a ProbLog program exploits two mapping functions which associate the probability Π_i and $1 - \Pi_i$ of a probabilistic fact with the positive and negative literals of the predicate, respectively. After the application of Clark's completion, the result may not be in Skolem normal form; thus, the techniques described above must be applied before executing WFOMC. The system WFOMC¹ solves the WFOMC problem by compiling the input theory into first-order d-DNNF diagrams [Darwiche, 2002; Chavira and Darwiche, 2008].

Example 84 (ProbLog program to Skolem normal form). *The translation of the ProbLog program of Example 79 into the WMC input format of the WFOMC system is*

```
predicate series1 1 1
predicate series2 1 1
predicate self 0.1 0.9
predicate at(P,A) 0.3 0.7
predicate z1 1 1
predicate s1 1 -1
predicate z2(P) 1 1
predicate s2(P) 1 -1
```

```
series v ! z1
!series v z1
z1 v !self
z1 v !attends(P)
z1 v s1
s1 v !self
s1 v !attends(P)
```

```
attends(P) v ! z2(P)
!attends(P) v z2(P)
z2(P) v !at(P,A)
z2(P) v s2(P)
s2(P) v !at(P,A)
```

¹<https://dtai.cs.kuleuven.be/software/wfomc>

Here, `predicate` is the mapping function for the probability values while z_1 and z_2 are Tseitin predicates and s_1 and s_2 are Skolem predicates.

6.5 Cyclic Logic Programs

LP^2 and aggregation parfactors, described in Sections 6.2 and 6.3, respectively, require a conversion from ProbLog to PFL for performing inference. The first step of this translation is the transformation of a ProbLog program into a BN with noisy-OR nodes. However, since BNs cannot have cycles, this conversion is not correct if the program is cyclic or *non-tight*, i.e., if the program contains positive cycles. A similar problem occurs with WFOMC: Clark’s completion [Clark, 1978] is correct only for acyclic logic programs.

Fages [1994] proved that if an LP program is acyclic, then the Herbrand models of its Clark’s completion [Clark, 1978] are minimal and coincide with the stable models of the original LP program. The consequence of this theoretical result is that, if the ProbLog program is acyclic, we can correctly convert it into a first-order theory by means of Clark’s completion.

To apply these techniques to cyclic programs, we need to remove positive loops. We could first apply the conversion proposed by Janhunen [2004] (also see Section 5.7) that converts normal logic programs to *atomic normal programs* then to clauses. An atomic normal program contains only rules of the form

$$a \leftarrow \sim c_1, \dots, \sim c_m.$$

where a and c_i are atoms. Such programs are tight and, as a consequence, it is possible to translate them into PFL programs and use Clark’s completion.

However, this conversion was proposed only for the case of ground LPs. Proposing a conversion for non-ground programs is an interesting direction for future work, especially if function symbols are allowed.

6.6 Comparison of the Approaches

Riguzzi et al. [2017a] experimentally compared LP^2 , C-FOVE with aggregation parfactors (C-FOVE-AP), and WFOMC on five problems:

- *workshops attributes* Milch et al. [2008];
- two different versions of *competing workshops* Milch et al. [2008];
- two different versions of Example 7 in Poole [2008], that we call *plates*.

According to Jaeger and Van den Broeck [2012], Van den Broeck [2011], function-free first-order logic with equality and two variables per formula ($2\text{-FFFOL}(=)$) is domain-liftable, i.e., the complexity of reasoning is polynomial in the domain size. All these problems fall in $2\text{-FFFOL}(=)$ and the experiments confirm that systems take polynomial time. However, WFOMC performs much better than the other systems, while LP^2 and C-FOVE-AP show approximately the same performance on all problems.

7

Approximate Inference

Approximate inference aims at computing the results of inference in an approximate way so that the process is cheaper than the exact computation of the results.

We can divide approaches for approximate inference into two groups: those that modify an exact inference algorithm and those based on sampling.

7.1 ProbLog1

ProbLog1 includes three approaches for approximately solving the EVID task. The first is based on iterative deepening and computes a lower and an upper bound for the probability of the query. The second instead approximates the probability of the query only from below using a fixed number of proofs. The third uses Monte Carlo sampling.

7.1.1 Iterative Deepening

In iterative deepening, the SLD tree is built only up to a certain depth [De Raedt et al., 2007; Kimmig et al., 2008]. Then two sets of explanations are built: K_l , encoding the successful proofs present in the tree, and K_u , encoding the successful and still open proofs present in the tree. The probability of K_l is a lower bound on the probability of the query, as some of the open derivations may succeed, while the probability of K_u is an upper bound on the probability of the query, as some of the open derivations may fail.

Example 85 (Path – ProbLog – iterative deepening). *Consider the program of Figure 7.1 which is a probabilistic version of the program of Example 1 and represents connectivity in the probabilistic graph of Figure 7.2.*

The query path(c, d) has the covering set of explanations

$$K = \{\{ce, ef, fd\}, \{cd\}\}$$

```

path(X, X).
path(X, Y) ← edge(X, Z), path(Z, Y).
0.8 :: edge(a, c).
0.7 :: edge(a, b).
0.8 :: edge(c, e).
0.6 :: edge(b, c).
0.9 :: edge(c, d).
0.625 :: edge(e, f).
0.8 :: edge(f, d).
    
```

Figure 7.1 Program for Example 85.

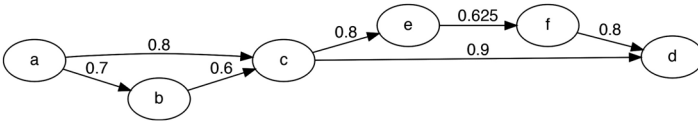


Figure 7.2 Probabilistic graph of Example 85.

where atomic choices $(f, \emptyset, 1)$ for facts of the form $f = \Pi :: edge(x, y)$ are represented as xy . K can be made pairwise incompatible as

$$K' = \{\{ce, ef, fd, \neg cd\}, \{cd\}\}$$

where $\neg cd$ indicates choice $(f, \emptyset, 0)$ for $f = 0.9 :: edge(c, d)$. The probability of the query is $P(\text{path}(c, d)) = 0.8 \cdot 0.625 \cdot 0.8 \cdot 0.1 + 0.9 = 0.94$.

For a depth limit of 4, we get the tree of Figure 7.3. This tree has one successful derivation, associated with the explanation $\kappa_1 = \{cd\}$, one failed derivation, and one derivation that is still open, the one ending with $\text{path}(f, d)$, that is associated with composite choice $\kappa_1 = \{ce, ef\}$,

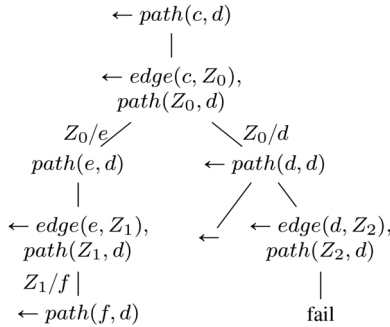


Figure 7.3 SLD tree up to depth 4 for the query $\text{path}(c, d)$ from the program of Example 85.

so $K_l = \{\kappa_1\}$ and $K_u = \{\kappa_1, \kappa_2\}$. We have $P(K_l) = 0.9$ and $P(K_u) = 0.95$ and $P(K_l) \leq P(\text{path}(c, d)) \leq P(K_u)$.

The iterative deepening algorithm of ProbLog1 takes as input an error bound ϵ , a depth bound d , and a query q . It constructs an SLD tree for q up to depth d . Then it builds sets of composite choices K_l and K_u and computes their probabilities. If the difference $P(K_u) - P(K_l)$ is smaller than the error bound ϵ , this means that a solution with a satisfying accuracy has been found and the interval $[P(K_l), P(K_u)]$ is returned. Otherwise, the depth bound is increased and a new SLD tree is built up to the new depth bound. This process is iterated until the difference $P(K_u) - P(K_l)$ becomes smaller than the error bound.

Instead of a depth bound, ProbLog1 can use a bound on the probability of the proof: when the probability of the explanation associated with a proof drops below a threshold, the proof is stopped. The threshold is reduced in the following iterations by multiplying it with a constant smaller than one.

7.1.2 k -best

The second approach for approximate inference in ProbLog1 uses a fixed number of proofs to obtain a lower bound of the probability of the query [Kimmig et al., 2008, 2011a]. Given an integer k , the best k proofs are found, corresponding to the set of best k explanations K_k , and the probability of K_k is used as an estimate of the probability of the query.

Best is here intended in terms of probability: an explanation is better than another if its probability is higher.

Example 86 (Path – ProbLog – k -best). *Consider the program of Example 85 with the query $\text{path}(a, d)$. This query has four explanations that are listed below together with their probabilities:*

$$\begin{aligned} \kappa_1 &= \{ac, cd\} & P(\kappa_1) &= 0.72 \\ \kappa_2 &= \{ab, bc, cd\} & P(\kappa_1) &= 0.378 \\ \kappa_3 &= \{ac, ce, ef, fd\} & P(\kappa_1) &= 0.32 \\ \kappa_4 &= \{ab, bc, ce, ef, fd\} & P(\kappa_1) &= 0.168 \end{aligned}$$

If $k = 1$, ProbLog1 considers only the best proof and $P(K_1) = 0.72$. For $k = 2$, ProbLog1 takes into account the best two explanations, $K_2 = \{\kappa_1, \kappa_2\}$. By making them pairwise incompatible, we get $K'_2 = \{\kappa_1, \{ab, bc, cd, \neg ac\}\}$ and $P(K'_2) = 0.72 + 0.378 \cdot 0.2 = 0.7956$. For $k = 3$, $K_3 = \{\kappa_1, \kappa_2, \kappa_3\}$ and $P(K_3) = 0.8276$. For $k = 4$, $K_4 = \{\kappa_1, \dots, \kappa_4\}$ and $P(K_4) = P(\text{path}(a, d)) = 0.83096$, the same for $k > 4$.

To perform k -best inference, ProbLog uses a branch-and-bound approach: the current best k explanations are kept and, when the probability of a derivation drops below the probability of the k -th best explanation, the derivation is cut. When a new explanation is found, it is inserted in the list of k -best explanations in order of probability, possibly removing the last one if the list already contains k explanations.

The algorithm returns a lower bound on the probability of the query, the larger the k the better the bound.

7.1.3 Monte Carlo

The Monte Carlo approach for approximate inference is based on the following procedure, to be repeated until convergence

1. Sample a world, by sampling each ground probabilistic fact in turn.
2. Check whether the query is true in the world.
3. Compute the probability \hat{p} of the query as the fraction of samples where the query is true.

Convergence is reached when the size of the confidence interval of \hat{p} drops below a user-defined threshold δ . In order to compute the confidence interval of \hat{p} , ProbLog1 uses the central limit theorem to approximate the binomial distribution with a normal distribution. Then the binomial proportion confidence interval is calculated as

$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

where n is the number of samples and $z_{1-\alpha/2}$ is the $1 - \alpha/2$ percentile of a standard normal distribution with $\alpha = 0.05$ usually. If the width of the interval is below the user-defined threshold δ , ProbLog1 stops and returns \hat{p} .

This estimate of the confidence interval is good for a sample size larger than 30 and if \hat{p} is not too close to 0 or 1. The normal approximation fails totally when the sample proportion is exactly zero or exactly one.

The above approach for generating samples, however, is not efficient on large programs, as proofs are often short while the generation of a world requires sampling many probabilistic facts. So ProbLog1 generates samples lazily by sampling probabilistic facts only when required by a proof. In fact, it is not necessary to sample facts not needed by a proof, as any value for them would do.

ProbLog1 performs a so-called *source-to-source transformation* of the program where probabilistic facts are transformed using the `term_expansion` mechanism of Prolog. For example, facts of the form

0.8 :: $edge(a, c)$.

0.7 :: $edge(a, b)$.

are transformed into

$edge(A, B) \leftarrow problog_edge(ID, A, B, LogProb)$,

$grounding_id(edge(A, B), ID, GroundID)$,

$add_to_proof(GroundID, LogProb)$.

$problog_edge(0, a, c, -0.09691)$.

$problog_edge(1, a, b, -0.15490)$.

where $problog_edge$ is a new predicate for the internal representation of facts for the predicate $edge/2$, $grounding_id/3$ is used in the case in which the probabilistic facts are not ground for obtaining a different identifier for each grounding, and $add_to_proof/2$ adds the fact to the current proof, stored in a global storage area. This approach is shared by all inference algorithms of ProbLog1.

The computation of \hat{p} is usually done after taking a user defined small number of samples n instead of after every sample, see Algorithm 6.

Algorithm 6 Function MONTECARLO: Monte Carlo algorithm of ProbLog1.

```

1: function MONTECARLO( $\mathcal{P}, q, n, \delta$ )
2:   Input: Program  $\mathcal{P}$ , query  $q$ , number of batch samples  $n$ , precision  $\delta$ 
3:   Output:  $P(q)$ 
4:   transform  $\mathcal{P}$ 
5:    $Samples \leftarrow 0$ 
6:    $TrueSamples \leftarrow 0$ 
7:   repeat
8:     for  $i = 1 \rightarrow n$  do
9:        $Samples \leftarrow Samples + 1$ 
10:      if SAMPLE( $q$ ) succeeds then
11:         $TrueSamples \leftarrow TrueSamples + 1$ 
12:      end if
13:    end for
14:     $\hat{p} \leftarrow \frac{TrueSamples}{Samples}$ 
15:    until  $2z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{Samples}} < \delta$ 
16:    return  $\hat{p}$ 
17: end function

```

The algorithm converges because the *Samples* variable is always increasing, and thus the condition $2z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{\text{Samples}}} < \delta$ in line 15 of Algorithm 6 will eventually become true, unless the query has probability 0 or 1.

The function `SAMPLE(q)` is implemented by asking the query over the transformed program. `ProbLog1` uses an array with an element for each ground probabilistic fact that stores one of three values: sampled true, sampled false, or not yet sampled. When a literal matching a probabilistic fact is called, `ProbLog1` first checks whether the fact was already sampled by looking at the array. If it wasn't sampled, `ProbLog1` samples it and stores the result in the array. Probabilistic facts that are non-ground in the program are treated differently: samples for groundings of these facts are stored in the internal database of the Prolog interpreter (YAP in the `ProbLog1` case) and the sampled value is retrieved when they are called. If no sample has been taken for a grounding, a sample is taken and recorded in the database. No position in the array is reserved for them since their grounding is not known at the start.

Approximate inference by sampling is also available in the `ProbLog2` system.

7.2 MCINTYRE

MCINTYRE (Monte Carlo INference wiTh Yap REcord) [Riguzzi, 2013] applies the Monte Carlo approach of `ProbLog1` to LPADs using the YAP internal database for storing all samples and using tabling for speeding up inference.

MCINTYRE first transforms the program and then queries the transformed program. The disjunctive clause

$$C_i = h_{i1} : \Pi_{i1} \vee \dots \vee h_{in} : \Pi_{in} : -b_{i1}, \dots, b_{im_i},$$

where the parameters sum to 1, is transformed into the set of clauses $MC(C_i) = \{MC(C_i, 1), \dots, MC(C_i, n_i)\}$:

$$MC(C_i, 1) = \quad h_{i1} : -b_{i1}, \dots, b_{im_i}, \\ \quad \quad \quad \text{sample_head}(\text{ParList}, i, VC, NH), NH = 1.$$

...

$$MC(C_i, n_i) = \quad h_{in_i} : -b_{i1}, \dots, b_{im_i}, \\ \quad \quad \quad \text{sample_head}(\text{ParList}, i, VC, NH), NH = n_i.$$

where *VC* is a list containing each variable appearing in C_i and *ParList* is $[\Pi_{i1}, \dots, \Pi_{in_i}]$. If the parameters do not sum up to 1, the last clause (the one

for *null*) is omitted. MCINTYRE creates a clause for each head and samples a head index at the end of the body with `sample_head/4`. If this index coincides with the head index, the derivation succeeds; otherwise, it fails. Thus, failure can occur either because one of the body literals fails or because the current clause is not part of the sample.

Example 87 (Epidemic – LPAD). *The following LPAD models the development of an epidemic or a pandemic and is similar to the ProbLog program of Example 66:*

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

Clause C_1 has two groundings, both with three atoms in the head, while clause C_2 has a single grounding with two atoms in the head, so overall there are $3 \times 3 \times 2 = 18$ worlds. The query *epidemic* is true in five of them and its probability is

$$\begin{aligned} P(\text{epidemic}) &= 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + \\ &\quad 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 \\ &= 0.588 \end{aligned}$$

Clause C_1 is transformed as

$$\begin{aligned} MC(C_1, 1) &= \text{epidemic} : -\text{flu}(X), \text{cold}, \\ &\quad \text{sample_head}([0.6, 0.3, 0.1], 1, [X], NH), NH = 1. \\ MC(C_1, 2) &= \text{pandemic} : -\text{flu}(X), \text{cold}, \\ &\quad \text{sample_head}([0.6, 0.3, 0.1], 1, [X], NH), NH = 2. \end{aligned}$$

The predicate `sample_head/4` samples an index from the head of a clause and uses the built-in YAP predicates `recorded/3` and `recorda/3` for, respectively, retrieving or adding an entry to the internal database.

Since `sample_head/4` is at the end of the body and since we assume the program to be range restricted, all the variables of the clause have been grounded when `sample_head/4` is called.

If the rule instantiation was already sampled, `sample_head/4` retrieves the head index with `recorded/3`; otherwise, it samples a head index with `sample/2`:

```
sample_head(_ParList, R, VC, NH) :-
    recorded(exp, (R, VC, NH), _) , ! .
sample_head(ParList, R, VC, NH) :-
    sample(ParList, NH) ,
    recorda(exp, (R, VC, NH), _) .
```

```

sample(ParList, HeadId) :-
    random(Prob),
    sample(ParList, 0, 0, Prob, HeadId).
sample([HeadProb|Tail], Index, Prev, Prob,
       HeadId) :-
    Succ is Index + 1,
    Next is Prev + HeadProb,
    (Prob =< Next ->
     HeadId = Index
    ;
     sample(Tail, Succ, Next, Prob, HeadId)
    ).

```

Tabling can be effectively used to avoid re-sampling the same atom. To take a sample from the program, MCINTYRE uses the following predicate

```

sample(Goal) :-
    abolish_all_tables,
    eraseall(exp),
    call(Goal).

```

For example, if the query is *epidemic*, resolution matches the goal with the head of clause $MC(C_1, 1)$. Suppose $flu(X)$ succeeds with $X/david$ and $cold$ succeeds as well. Then

$$sample_head([0.6, 0.3, 0.1], 1, [david], NH)$$

is called. Since clause 1 with X replaced by *david* was not yet sampled, a number between 1 and 3 is sampled according to the distribution $[0.6, 0.3, 0.1]$ and stored in NH . If $NH = 1$, the derivation succeeds and the goal is true in the sample, if $NH = 2$ or $NH = 3$, then the derivation fails and backtracking is performed. This involves finding the solution $X/robert$ for $flu(X)$. *cold* was sampled as true before, so it succeeds again. The

$$sample_head([0.6, 0.3, 0.1], 1, [robert], NH)$$

is called to take another sample.

Differently from ProbLog1, MCINTYRE takes into account the validity of the binomial proportion confidence interval. The normal approximation is good for a sample size larger than 30 and if \hat{p} is not too close to 0 or 1. Empirically, it has been observed that the normal approximation works well

as long as $Sample \cdot \hat{p} > 5$ and $Sample \cdot (1 - \hat{p}) > 5$ [Ryan, 2007]. Thus, MCINTYRE changes the condition in line 15 of Algorithm 6 to

$$2z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{Samples}} < \delta \wedge Samples \cdot \hat{p} > 5 \wedge Samples \cdot (1-\hat{p}) > 5$$

Recent versions of MCINTYRE for SWI-Prolog (included in the `cplint` suite) use dynamic clauses for storing samples, as in SWI-Prolog these are faster. `sample_head/4` is then defined as:

```
sample_head(R, VC, _HeadList, N) :-
    sampled(R, VC, N), !.

sample_head(R, VC, HeadList, N) :-
    sample(HeadList, N),
    assertz(sampled(R, VC, N)).
```

Monte Carlo sampling is attractive for the simplicity of its implementation and because the estimate can be improved as more time is available, making it an *anytime algorithm*.

7.3 Approximate Inference for Queries with an Infinite Number of Explanations

Monte Carlo inference can also be used for programs with function symbols, in which goals may have an infinite number of possibly infinite explanations and exact inference may loop. In fact, a sample of a query corresponds naturally to an explanation. The probability of taking that sample is the same as the probability of the corresponding explanation. The risk is that of incurring in an infinite explanation. But infinite explanations have probability zero, so the probability that the computation goes down such a path and does not terminate is zero as well. As a consequence, Monte Carlo inference can be used on programs with an infinite number of possibly infinite explanations.

Similarly, iterative deepening can also avoid infinite loops as the proof tree is built only up to a certain point. If the bound is on the depth, computation will eventually stop because the depth bound will be exceeded. If the bound is on the probability, it will eventually be exceeded as well, as the probability of an explanation goes to zero as more choices are added.

For an example of Monte Carlo inference on a program with an infinite set of explanations, see Section 11.11.

7.4 Conditional Approximate Inference

Monte Carlo inference also provides smart algorithms for computing the probability of a query given evidence (COND task): rejection sampling or Metropolis-Hastings Markov Chain Monte Carlo (MCMC).

In rejection sampling [Von Neumann, 1951], the evidence is first queried and, if it is successful, the query is asked in the same sample; otherwise, the sample is discarded. Rejection sampling is available both in `cpaint` and in `ProbLog2`.

In Metropolis-Hastings MCMC, a Markov chain is built by taking an initial sample and by generating successor samples, see [Koller and Friedman, 2009] for a description of the general algorithm.

Nampally and Ramakrishnan [2014] developed a version of MCMC specific to PLP. In their algorithm, the initial sample is built by randomly sampling choices so that the evidence is true. A successor sample is obtained by deleting a fixed number (*lag*) of sampled probabilistic choices. Then the evidence is queried again by sampling starting with the undeleted choices. If the evidence succeeds, the query is then also asked by sampling. The query sample is accepted with a probability of

$$\min \left\{ 1, \frac{N_{i-1}}{N_i} \right\}$$

where N_{i-1} is the number of choices sampled in the previous sample and N_i is the number of choices sampled in the current sample. The number of successes of the query is increased by 1 if the query succeeded in the last accepted sample. The final probability is given by the number of successes over the total number of samples. Nampally and Ramakrishnan [2014] prove that this is a valid Metropolis-Hastings MCMC algorithm if *lag* is equal to 1.

Metropolis-Hastings MCMC is also implemented in `cpaint` [Alberti et al., 2017]. Since the proof of the validity of the algorithm in [Nampally and Ramakrishnan, 2014] also holds when forgetting more than one sampled choice, *lag* is user-defined in `cpaint`.

Algorithm 7 shows the procedure. Function `INITIALSAMPLE` returns a composite choice containing the choices sampled for proving the evidence. Function `SAMPLE` takes a goal and a composite choice as input and samples the goal returning a couple formed by the result of sampling (true or false) and the set of sampled choices extending the input composite choice. Function `RESAMPLE(κ , lag)` deletes *lag* choices from κ . In [Nampally and Ramakrishnan, 2014], *lag* is always 1. Function `ACCEPT(κ_{i-1} , κ_i)` decides whether to accept sample κ_i .

Algorithm 7 Function MCMC: Metropolis-Hastings MCMC algorithm.

```

1: function MCMC( $\mathcal{P}, q, Samples, lag$ )
2:   Input: Program  $\mathcal{P}$ , query  $q$ , number of samples  $Samples$ , number of choices to delete
        $lag$ 
3:   Output:  $P(q|e)$ 
4:    $TrueSamples \leftarrow 0$ 
5:    $\kappa_0 \leftarrow \text{INITIALSAMPLE}(e)$ 
6:    $(r_q, \kappa) \leftarrow \text{SAMPLE}(q, \kappa_0)$ 
7:   for  $i = 1 \rightarrow Samples$  do
8:      $\kappa' \leftarrow \text{RESAMPLE}(\kappa, lag)$ 
9:      $(r_e, \kappa_e) \leftarrow \text{SAMPLE}(e, \kappa')$ 
10:    if  $r_e = \text{true}$  then
11:       $(r'_q, \kappa_q) \leftarrow \text{SAMPLE}(q, \kappa_e)$ 
12:      if  $\text{ACCEPT}(\kappa, \kappa_q)$  then
13:         $\kappa \leftarrow \kappa_q$ 
14:         $r_q \leftarrow r'_q$ 
15:      end if
16:    end if
17:    if  $r_q = \text{true}$  then
18:       $TrueSamples \leftarrow TrueSamples + 1$ 
19:    end if
20:  end for
21:   $\hat{p} \leftarrow \frac{TrueSamples}{Samples}$ 
22:  return  $\hat{p}$ 
23: end function

```

Function INITIALSAMPLE builds the initial sample with a meta-interpreter (see Section 1.3) that starts with the goal and randomizes the order in which clauses are used for resolution during the search so that the initial sample is unbiased. This is achieved by collecting all the clauses that match a subgoal and trying them in random order. Then the goal is queried using regular sampling.

7.5 Approximate Inference by Sampling for Hybrid Programs

Monte Carlo inference also has the attractive feature that it can be used almost directly for approximate inference for hybrid programs. For example, to handle a hybrid clause of the form

$$C_i = g(X, Y) : \text{gaussian}(Y, 0, 1) \leftarrow \text{object}(X).$$

MCINTYRE transforms it into [Riguzzi et al., 2016a; Alberti et al., 2017]:

$$g(X, Y) \leftarrow \text{object}(X), \text{sample_gauss}(i, [X], 0, 1, Y).$$

Samples for continuous random variables are stored using asserts as for discrete variables. In fact, predicate *sample_gauss/4* is defined by

```
sample_gauss(R, VC, _Mean, _Variance, S) :-
    sampled(R, VC, S), !.
```

```
sample_gauss(R, VC, Mean, Variance, S) :-
    gauss(Mean, Variance, S),
    assertz(sampled(R, VC, S)).
```

where *gauss(Mean, Variance, S)* returns in *S* a value sampled from a Gaussian distribution with parameters *Mean* and *Variance*.

Monte Carlo inference for hybrid programs derives his correctness from the stochastic T_P operator: a clause that is ground except for the continuous random variable defined in the head defines a sampling process that extracts a sample of the continuous variables according to the distribution and parameters specified in the head. Since programs are range restricted, when the sampling predicate is called, all variables in the clause are ground except for the one defined by the head, so the T_P operator can be applied to the clause to sample a value for the defined variable.

Monte Carlo inference is the most common inference approach also for imperative or functional probabilistic programming languages, where a sample of the output of a program is taken by executing the program and generating samples when a probabilistic primitive is called. In probabilistic programming usually memoing is not used, so new samples are taken each time a probabilistic primitive is encountered.

Conditional inference can be performed in a similar way by using rejection sampling or Metropolis-Hastings MCMC, unless the evidence is on ground atoms that have continuous values as arguments. In this case, rejection sampling or Metropolis-Hastings cannot be used, as the probability of the evidence is 0. However, the conditional probability of the query given the evidence may still be defined, see Section 1.5. In this case, *likelihood weighting* [Nitti et al., 2016] can be used.

For each sample to be taken, likelihood weighting samples the query and then assigns a weight to the sample on the basis of evidence. The weight is computed by deriving the evidence backward in the same sample of the query starting with a weight of one: each time a choice should be taken or a continuous variable sampled, if the choice/variable has already been sampled,

the current weight is multiplied by the probability of the choice/by the density value of the continuous variable.

Then the probability of the query is computed as the sum of the weights of the samples where the query is true divided by the total sum of the weights of the samples. This technique is useful also for non-hybrid programs as samples are never rejected, so sampling can be faster.

Likelihood weighting in `cpInt` [Alberti et al., 2017; Nguembang Fadja and Riguzzi, 2017] uses a meta-interpreter that randomizes the choice of clauses when more than one resolves with the goal, in order to obtain an unbiased sample. This meta-interpreter is similar to the one used to generate the first sample in Metropolis-Hastings.

Then a different meta-interpreter is used to evaluate the weight of the sample. This meta-interpreter starts with the evidence as the query and a weight of 1. Each time the meta-interpreter encounters a probabilistic choice, it first checks whether a value has already been sampled. If so, it computes the probability/density of the sampled value and multiplies the weight by it. If the value has not been sampled, it takes a sample and records it, leaving the weight unchanged. In this way, each sample of the query is associated with a weight that reflects the influence of evidence.

In some cases, likelihood weighting encounters numerical problems, as the weights of samples may go rapidly to very small numbers that can be rounded to 0 by floating point arithmetic. This happens, for example, for dynamic models, where predicates depend on time and we have evidence for many time points. In these cases, *particle filtering* can be used [Nitti et al., 2016], which periodically resamples the individual samples/particles so that their weight is reset to 1.

In particle filtering, the evidence is a list of literals. A number n of samples of the query is taken that are weighted by the likelihood of the first element of the evidence list. Each sample constitutes a particle and the sampled random variables are stored away.

After weighting, n particles are resampled with replacement with a probability proportional to their weight. Specifically, the weights of the previous n particles are normalized. Let w_i be the normalized weight of particle s_i . Each of the new n particles is sampled from the set of previous particles with particle s_i selected with probability w_i . After each sample, the sampled particle is replaced back into the set so that the same particle can be sampled repeatedly.

After resampling, the next element of the evidence is considered. A new weight for each particle is computed on the basis of the new evidence element and the process is repeated until the last evidence element.

7.6 Approximate Inference with Bounded Error for Hybrid Programs

Michels et al. [2016] present the Iterative Hybrid Probabilistic Model Counting (IHPMC) algorithm for computing bounds on queries to hybrid programs. They consider both the EVID and COND tasks for the PCLP language (see Section 4.5). IHPMC builds trees that split the variables domains and builds them to an increasing depth in order to achieve the desired accuracy.

A Hybrid Probability Tree (HPT) is a binary tree where each node n is associated with a propositional formula φ_n and a range denoted by $range(n, X)$ for each random variable X . For the root node r , the range of each variable is its whole range: $range(r, X) = Range_X$. Each non-leaf node n splits the range of a random variable into two parts and the propositional formula of each child is obtained from that for the node by conditioning on the split made. Since the variables may be continuous, the range of a random variable can be split several times along the same tree branch. If the children of n are $\{c_1, c_2\}$, each edge $n \rightarrow c_i, i \in \{1, 2\}$ is associated with a range τ_{ni} for a random variable Y such that $\tau_{n1} \cup \tau_{n2} = range(n, Y)$ and $\tau_{n1} \cap \tau_{n2} = \emptyset$. Moreover, $range(c_i, Y) = \tau_{ni}$ and $range(c_i, Z) = range(n, Z)$ if $Y \neq Z$, with $i \in \{1, 2\}$.

Then the formula φ_{c_i} associated to c_i is obtained from φ_n by imposing a restriction on the range on Y and the formula is simplified if some primitive constraints can be replaced by \top or \perp . Each edge $n \rightarrow c_i$ is associated with a probability p_{ni} such that $p_{ni} = P(Y \in \tau_{ni} | Y \in range(n, Y))$. Leaf nodes l are those where $\varphi_l = \top$ or $\varphi_l = \perp$.

Given an HPT, the probability of the event represented by the formula associated with the root can be computed using Algorithm 4 for computing the probability of a BDD.

Example 88 (Machine diagnosis problem [Michels et al., 2016]). *Consider a diagnosis problem where a machine fails if the temperature is above a threshold. If cooling fails ($noc = true$), then the threshold is lower. This problem can be modeled with the PCLP program:*

```
fail ← t > 30.0
fail ← t > 20.0, noc = true
```

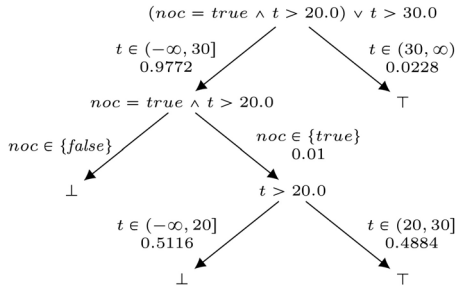


Figure 7.4 HPT for Example 88. From [Michels et al., 2016].

$t \sim \text{gaussian}(20.0, 50.0)$

$\text{noc} \sim \{0.01 : \text{true}.0.99 : \text{false}\}$

The event that the machine fails is then represented by the formula

$$(\text{noc} = \text{true} \wedge t > 20.0) \vee t > 30.0$$

The HPT for this formula is shown in Figure 7.4.

The probability of fails is then

$$P(\text{fails}) = 0.9772 \cdot 0.01 \cdot 0.4884 + 0.0228 \approx 0.0276$$

In Example 88, we were able to compute the probability of the query exactly. In general, this may not be possible because the query event may not be representable using hyperrectangles over the random variables. In this case, the propositional formulas may never simplify to \perp or \top . However, we can consider Partially evaluated Hybrid Probability Trees (PHPTs), HPTs where not all leaves are \perp or \top . From a PHPT, we can obtain a lower and an upper bound on the probability of the query: the \top leaves in the tree contribute to the lower bound $\underline{P}(q)$ and all the leaves of the tree except the \perp ones contribute to the upper bound $\overline{P}(q)$.

Example 89 (Machine diagnosis problem – approximate inference – [Michels et al., 2016]). Consider the program:

$\text{fail} \leftarrow t > l$

$t \sim \text{gaussian}(20.0, 50.0)$

$l \sim \text{gaussian}(30.0, 50.0)$

The event that the machine fails is then represented by the formula $t > l$ and the PHPT for this formula is shown in Figure 7.5.

The lower bound of the probability of fails is then

$$\underline{P}(\text{fails}) = 0.9772 \cdot 0.5$$

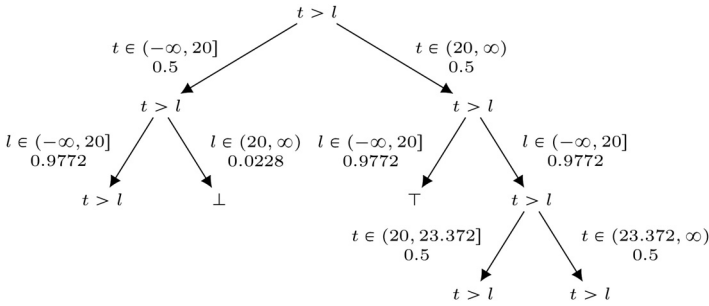


Figure 7.5 PHPT for Example 89. From [Michels et al., 2016].

Since the only \perp leaf has probability $0.0228 \cdot 0.5$, then

$$\overline{P}(\text{fails}) = 1 - 0.0228 \cdot 0.5$$

Given a precision ϵ , IHPMC builds a PHPT such that $P(q) - \underline{P}(q) \leq \epsilon$ and $\overline{P}(q) - P(q) \leq \epsilon$. To do so, the tree must be built to a sufficient depth.

To compute bounds on conditional probabilities, IHPMC needs to compute bounds for $q \wedge e$ and $\neg q \wedge e$ where q is the query and e the evidence. IHPMC computes the bounds using two PHPTs. Arbitrary precision can still be achieved by building the trees to a sufficient depth.

When building trees, IHPMC uses heuristics for selecting the next node to expand, the variable to be used for splitting, and the partitioning of the range of the variable. IHPMC expands the leaf node with the highest probability mass. Variables occurring more frequently in the formula are preferred for splitting, unless a different choice may eliminate a primitive constraint. For partitioning continuous variables, split points that can lead to a simplification of the formula are selected. If there are no such points, a partitioning is chosen that splits evenly the probability of the node.

Weighted model integration [Belle et al., 2015b,a, 2016; Morettin et al., 2017] is a recent approach that generalizes WMC to hybrid programs. It can deal with general continuous distributions by approximating them with piecewise polynomials. While approximations can get arbitrarily close to the exact distributions, the method does not provide bounds on the error of the result.

7.7 *k*-Optimal

In *k*-best, the set of proofs can be highly redundant with respect to each other. *k*-optimal [Renkens et al., 2012] improves on *k*-best for definite clauses by trying to find the set of *k* explanations $K = \{\kappa_1, \dots, \kappa_k\}$ of the query q that lead to the largest probability $P(q)$, in order to obtain the best possible lower bound given the limit on the number of explanations.

k-optimal follows a greedy procedure shown in Algorithm 8. The optimization in line 4 is performed with an algorithm similar to 1-best: a branch-and-bound search is performed where, instead of evaluating the current partial explanation κ using its probability, the value $P(K \cup \{\kappa\}) - P(K)$ is used.

Algorithm 8 Function K-OPTIMAL: *k*-optimal algorithm.

```

1: function K-OPTIMAL( $\phi_r, \phi_q, \text{maxTime}$ )
2:    $K \leftarrow \emptyset$ 
3:   for  $i = 1 \rightarrow k$  do
4:      $K \leftarrow K \cup \arg \max_{\kappa \text{ is an explanation}} P(K \cup \{\kappa\})$ 
5:   end for
6:   return  $K$ 
7: end function

```

In order to efficiently compute $P(K \cup \{\kappa\})$, *k*-optimal uses compilation to BDDs. Instead of building the BDD for $K \cup \{\kappa\}$ from scratch, *k*-optimal uses a smarter approach. Let *dnf* represent the DNF formula for K and let $f_1 \wedge \dots \wedge f_n$ represent the Boolean formula for κ , where the f_i s are Boolean variables for ground probabilistic facts. Then

$$\begin{aligned}
 P(f_1 \wedge \dots \wedge f_n \vee \text{dnf}) &= P(f_1 \wedge \dots \wedge f_n) + \\
 &\quad P(\neg f_1 \wedge \text{dnf}) + \\
 &\quad P(f_1 \wedge \neg f_2 \wedge \text{dnf}) + \dots \\
 &\quad P(f_1 \wedge \dots \wedge f_{n-1} \wedge \neg f_n \wedge \text{dnf})
 \end{aligned}$$

Since probabilistic facts are independent, $P(f_1 \wedge \dots \wedge f_n)$ can be easily computed as $P(f_1) \cdot \dots \cdot P(f_n)$. The other terms become

$$\begin{aligned}
 P(f_1 \wedge \dots \wedge f_{i-1} \wedge \neg f_i \wedge \text{dnf}) &= P(f_1) \cdot \dots \cdot P(f_{i-1}) \cdot (1 - P(f_i)) \cdot \\
 &\quad P(\text{dnf} | f_1 \wedge \dots \wedge f_{i-1} \wedge \neg f_i)
 \end{aligned}$$

The factor $P(\text{dnf} | f_1 \wedge \dots \wedge f_{i-1} \wedge \neg f_i)$ can be computed cheaply if the BDD for *dnf* is available: we can apply function PROB of Algorithm 4 by assuming that $P(f_j) = 1$ for the conditional facts $j < i$ and $P(f_i) = 0$.

So, at the beginning of a search iteration, K is compiled to a BDD and $P(K \cup \{\kappa\})$ is computed for each node of the SLD tree for the query q , where κ is the composite choice corresponding to the node, representing a possibly partial explanation. If κ has n elements, n conditional probabilities must be computed.

However, when the probability $P(f_1 \wedge \dots \wedge f_n \wedge f_{n+1} \vee dnf)$ for the partial proof $f_1 \wedge \dots \wedge f_n \wedge f_{n+1}$ must be computed, the probability $P(f_1 \wedge \dots \wedge f_n \vee dnf)$ was computed in the parent of the current node. Since

$$P(f_1 \wedge \dots \wedge f_n \wedge f_{n+1} \vee dnf) = P(f_1 \wedge \dots \wedge f_n \vee dnf) + P(f_1 \wedge \dots \wedge f_n \wedge \neg f_{n+1} \wedge dnf)$$

then only $P(f_1 \wedge \dots \wedge f_n \wedge \neg f_{n+1} \wedge dnf)$ must be computed.

In practice, Renkens et al. [2012] observe that computing $P(K \cup \{\kappa\})$ for each node of the SLD tree is still too costly because partial proofs have also to be considered and these may lead to dead ends. They found experimentally that using the bound $P(\kappa)$ of k -best for pruning incomplete proofs and computing $P(K \cup \{\kappa\})$ only when a complete proof is found provides better performance. This works because $P(\kappa)$ is an upper bound on $P(K \cup \{\kappa\}) - P(K)$, so cutting a branch because $P(\kappa)$ has become smaller than $P(K \cup \{\kappa'\}) - P(K)$ for the best explanation κ' found so far does not prune good solutions.

However, this approach performs less pruning, as it is based on an upper bound, and the computation of $P(K \cup \{\kappa\})$ at the parent of an SLD node of a complete proof is no longer available.

k -optimal, as k -best, still suffers from the problem that k is set beforehand and fixed, so it may happen that, of the k proofs, many provide only very small contributions and a lower value of k could have been used. k - θ -optimal puts a threshold θ on the added proof probability: k -optimal is stopped before k proofs are found if no more proof has an added probability $P(K \cup \{\kappa\}) - P(K)$ larger than θ . This is achieved by setting the bound to θ at the beginning of each iteration of k -optimal.

Renkens et al. [2012] prove that the k -optimal optimization problem is NP-hard. They also show that the proposed greedy algorithm achieves an approximation which is not worse than $1 - \frac{1}{e}$ times the probability of the optimal solution.

The experiments in [Renkens et al., 2012] performed on biological graphs show that k -best is about an order of magnitude faster than k -optimal but that k -optimal obtains better bounds especially when k is low compared to the number of available proofs.

7.8 Explanation-Based Approximate Weighted Model Counting

Renkens et al. [2014] solve EVID approximately by computing lower and upper bounds of the probability of evidence from ProbLog programs. The approach is based on computing explanations for evidence e one by one, where an explanation here is represented as a conjunction of Boolean variables representing ground probabilistic facts. So an explanation

$$\kappa = \{(f_1, \theta, k_1), \dots, (f_n, \emptyset, k_n)\}$$

for ground probabilistic facts $\{f_1, \dots, f_n\}$ is represented as

$$exp_\kappa = \bigwedge_{(f, \emptyset, 1) \in \kappa} \lambda_f \wedge \bigwedge_{(f, \emptyset, 0) \in \kappa} \neg \lambda_f$$

where λ_f is the Boolean variable associated with fact f .

An explanation exp is such that $\phi_r \wedge exp \models \phi_e$, where ϕ_r and ϕ_e are the propositional formulas representing the rules and the evidence, respectively, computed as in ProbLog2 (see Section 5.7).

Renkens et al. [2014] show that, given any set of explanations

$$\{exp_1, \dots, exp_m\},$$

it holds that

$$WMC_{\mathbf{V}}(\phi \wedge (exp_1 \vee \dots \vee exp_m)) = WMC_{\mathbf{E}}(exp_1 \vee \dots \vee exp_m)$$

where $\phi = \phi_r \wedge \phi_e$ and \mathbf{V} is the set of all variables of the programs, i.e., those for probabilistic facts and those of the head of clauses in the grounding of the program, while \mathbf{E} is the set of variables of the program for probabilistic facts only.

Given two weighted formulas ψ and ξ over the same set of variables \mathbf{V} , we have that $WMC_{\mathbf{V}}(\psi) \geq WMC_{\mathbf{V}}(\psi \wedge \xi)$ as the models of $\psi \wedge \xi$ are a subset of those of ψ . So $WMC_{\mathbf{V}}(\phi \wedge (exp_1 \vee \dots \vee exp_m))$ is a lower bound for $WMC_{\mathbf{V}}(\phi)$ and tends toward it as the number explanations increases, as each explanation encodes a set of worlds. Since the number of explanations is finite, when considering all explanations, the two counts will be equal.

Moreover, we can compute $WMC_{\mathbf{V}}(\phi \wedge (exp_1 \vee \dots \vee exp_m))$ more quickly by computing $WMC_{\mathbf{E}}(exp_1 \vee \dots \vee exp_m)$ because it has fewer variables. This leads to Algorithm 9 for computing a lower bound of the evidence, where function NEXTEXPL returns a new explanation. The algorithm is anytime, we can stop it at any time still obtaining a lower bound of $P(e)$.

Algorithm 9 Function AWMC: Approximate WMC for computing a lower bound of $P(e)$.

```

1: function AWMC( $\phi_r, \phi_e, \text{maxTime}$ )
2:    $\psi \leftarrow 0$ 
3:   while  $\text{time} < \text{maxTime}$  do
4:      $\text{exp} \leftarrow \text{NEXTEXPL}(\phi_r \wedge \phi_e)$ 
5:      $\psi \leftarrow \psi \vee \text{exp}$ 
6:   end while
7:   return  $\text{WMC}_{\mathcal{V}(\psi)}(\psi)$ 
8: end function

```

NEXTEXPL looks for the next best explanation, i.e., the one with maximal probability (or WMC). This is done by solving a weighted MAX-SAT problem: given a CNF formula with non-negative weights assigned to clauses, find assignments for the variables that minimize the sum of the weights of the violated clauses. An appropriate weighted CNF formula built over an extended set of variables is passed to a weighted MAX-SAT solver that returns an assignment for all the variables. An explanation is built from the assignment. To ensure that the same explanation is not found every time, a new clause excluding it is added to the CNF formula for every found explanation.

An upper bound of $P(e)$ can be computed by observing that, for ProbLog, $\text{WMC}(\phi_r) = 1$, because the variables for probabilistic facts can take any combination of values, the weights for their literals sum to 1 (represent a probability distribution) and the weight for derived literals (those of atoms appearing in the head of clauses) are all 1, so they don't influence the weight of worlds. Therefore

$$\text{WMC}(\phi_r \wedge \phi_e) = \text{WMC}(\phi_r) - \text{WMC}(\phi_r \wedge \neg\phi_e) = 1 - \text{WMC}(\phi_r \wedge \neg\phi_e)$$

As a consequence, if we compute a lower bound on $\text{WMC}(\phi_r \wedge \neg\phi_e)$, we can derive an upper bound on $\text{WMC}(\phi_r \wedge \phi_e)$. The lower bound on $\text{WMC}(\phi_r \wedge \neg\phi_e)$ is computed as for $\text{WMC}(\phi_r \wedge \phi_e)$, by looking for explanations for $\phi_r \wedge \neg\phi_e$.

This leads to Algorithm 10 that, at each iteration, updates the bound that at the previous iteration had the largest change in value. The algorithm is anytime: at any time point, *low* and *up* are the lower and upper bounds of $P(e)$, respectively.

Algorithm 10 Function AWMC: Approximate WMC for computing lower and upper bounds of $P(e)$.

```

1: function AWMC( $\phi_r, \phi_e, maxTime$ )
2:    $improveTop \leftarrow 0.5$ 
3:    $improveBot \leftarrow 0.5$ 
4:    $top \leftarrow 1$ 
5:    $bot \leftarrow 0$ 
6:    $up \leftarrow 1.0$ 
7:    $low \leftarrow 0.0$ 
8:   while  $time < maxTime$  do
9:     if  $improveTop > improveBot$  then
10:       $exp \leftarrow \text{NEXTEXPL}(\phi_r \wedge \neg\phi_e)$ 
11:       $next \leftarrow \text{WMC}(top \wedge \neg exp)$ 
12:       $improveTop \leftarrow up - next$ 
13:       $top \leftarrow top \wedge \neg exp$ 
14:       $up \leftarrow next$ 
15:     else
16:       $exp \leftarrow \text{NEXTEXPL}(\phi_r \wedge \phi_e)$ 
17:       $next \leftarrow \text{WMC}(bot \vee exp)$ 
18:       $improveBot \leftarrow next - low$ 
19:       $bot \leftarrow bot \vee exp$ 
20:       $low \leftarrow next$ 
21:     end if
22:   end while
23:   return [ $low, up$ ]
24: end function

```

7.9 Approximate Inference with T_P -compilation

T_P compilation [Vlasselaer et al., 2015, 2016] discussed in Section 5.8 can be used to perform approximate CONDATOMS inference by computing a lower and an upper bound of the probabilities of query atoms, similarly to iterative deepening of Section 7.1.1.

Vlasselaer et al. [2016] show that, for each iteration i of application of T_{CP} , if λ_a^i is the formula associated with atom a in the result of $T_{CP} \uparrow i$, then $WMC(\lambda_a^i)$ is a lower bound on $P(a)$. So T_P compilation is an anytime algorithm for approximate inference: at any time, the algorithm provides a lower bound of the probability of each atom.

Moreover, T_{CP} is applied using the one atom at a time approach where the atom to evaluate is selected using a heuristic that is

- proportional to the increase in the probability of the atom;
- inversely proportional to the complexity increase of the SDD for the atom;
- proportional to the importance of the atom for the query, computed as the inverse of the minimal depth of the atom in the SLD trees for each of the queries of interest.

An upper bound for definite programs is instead computed by selecting a subset \mathcal{F}' of the facts \mathcal{F} of the program and by assigning them the value true, which is achieved by conjoining each λ_a with $\lambda_{\mathcal{F}'} = \bigwedge_{f \in \mathcal{F}'} \lambda_f$. If we then compute the fixpoint, we obtain an upper bound: $WMC(\lambda_a^\infty) \geq P(a)$. Moreover, conjoining with $\lambda_{\mathcal{F}'}$ simplifies formulas and so also compilation.

The subset \mathcal{F}' is selected by considering the minimal depth of each fact in the SLD trees for each of the queries and by inserting into \mathcal{F}' only the facts with a minimal depth smaller than a constant d . This is done to make sure that the query depends on at least one probabilistic fact and the upper bound is smaller than 1.

While the lower bound is also valid for normal programs, the upper bound can be used only for definite programs.

7.10 DISTR and EXP Tasks

Monte Carlo inference can also be used to solve DISTR and EXP tasks, i.e., computing probability distributions or expectations over arguments of queries. In this case, the query contains one or more variables. In the DISTR task, we record the values of these arguments in successful samples of the query. If the program is range restricted, queries succeed with all arguments instantiated, so these values are guaranteed to exist. For unconditional inference or conditional inference with rejection sampling or Metropolis-Hastings, the result is a list of terms, one for each sample. For likelihood weighting, the results is a list of weighted terms, where the weight of each term is the sample weight.

From these lists, approximations of probability distributions or probability densities can be built, depending on the type of values, discrete or continuous, respectively. For an unweighted list of discrete values, the probability of each value is the number of occurrences of that value in the list, divided by the total number of values. For a weighted list of discrete values, the probability of each value is the sum of the weights of each occurrence of the value in the list, divided by the sum of all the weights.

For an unweighted list of continuous values, a line plot of the probability density function can be drawn by dividing the domain of the variable in a number of intervals or bins. The function then has a point for each interval, whose y value is the number of values in the list that fall in the interval, divided by the total number of values. For a weighted list of continuous values, the y value is the sum of the weights of each value in the list that falls in the interval, divided by the sum of all the weights.

Note that, if likelihood weighting is used, the set of samples without the weight can be interpreted as the density of the variable prior to the observation of the evidence. So we can draw a plot of the density before and after observing the evidence.

The EXP task can be solved by first solving the DISTR task, where we collect a list of numerical values, possibly weighted. Then the required value can be computed as a (weighted) mean. For unweighted samples, this is given by

$$\mathbf{E}(q|e) = \frac{\sum_{i=1}^n v_i}{n}$$

where $[v_1, \dots, v_n]$ is the list of values. For weighted samples, it is given by

$$\mathbf{E}(q|e) = \frac{\sum_{i=1}^n w_i \cdot v_i}{\sum_{i=1}^n w_i}$$

where $[(v_1, w_1), \dots, (v_n, w_n)]$ is the list of weighted values, with w_i the weight.

`cplint` [Alberti et al., 2017; Nguembang Fadja and Riguzzi, 2017] offers functions for performing both DISTR and EXP using sampling, rejection sampling, Metropolis-Hastings, likelihood weighting, and particle filtering, see Section 11.1.

Notice that a query may return more than one value for an output argument in a given world. The query returns a single value for each sample only if the query predicate is determinate in each world. A predicate is *determinate* if, given values for input arguments of a query over that predicate, there is a single value for output arguments that makes the query true. The user should be aware of this and write the program so that predicates are determinate, or otherwise consider only the first value for a sample. Or he may be interested in all possible values for the output argument in a sample, in which case a call to `findall/3` should be wrapped around the query and the resulting list of sampled values will be a list of lists of values.

If the program is not determinate, the user may be interested in a sampling process where first the world is sampled, and then the value of the output

argument is sampled uniformly from the set of values that make the query true. In this case, to ensure uniformity, a meta-interpreter that randomizes the choice of clauses for resolution should be used, such as the one used by `cplint` in likelihood weighting.

Programs satisfying the exclusive-or assumption, see Chapter 5, are determinate, because, in program satisfying this assumption, clauses sharing an atom in the head are mutually exclusive, i.e., in each world, the body of at most one clause is true. In fact, the semantics PRISM, where this assumption is made, can also be seen as defining a probability distribution over the values of output arguments.

Some probabilistic logic languages, such as SLPs (see Section 2.11.1), directly define probability distributions over arguments rather than probability distributions over truth values of ground atoms. Inference in such programs can be simulated with programs under the DS by solving the DISTR task.

Example 90 (Generative model). *The following program¹ encodes the model from [Goodman and Tenenbaum, 2018] for generating random functions:*

```
eval(X,Y) :- random_fn(X,0,F), Y is F.
op(+):0.5; op(-):0.5.
random_fn(X,L,F) :- comb(L), random_fn(X,l(L),F1),
    random_fn(X,r(L),F2), op(Op), F=..[Op,F1,F2].
random_fn(X,L,F) :- \+comb(L),base_random_fn(X,L,F).
comb(_):0.3.
base_random_fn(X,L,X) :- identity(L).
base_random_fn(_,L,C) :- \+identity(L),
    random_const(L,C).

identity(_):0.5.
random_const(_,C):discrete(C,[0:0.1,1:0.1,2:0.1,
    3:0.1,4:0.1,5:0.1,6:0.1,7:0.1,8:0.1,9:0.1]).
```

A random function is either an operator (“+” or “−”) applied to two random functions or a base random function. A base random function is either an identity or a constant drawn uniformly from the integers 0, . . . , 9.

You may be interested in the distribution of all possible output values of the random function with input 2 given that the function outputs 3 for input 1.

If we take 1000 samples with Metropolis-Hastings, we may get the bar graph of the frequencies of the sampled values shown in Figure 7.6. Since each world of the program is determinate, there is a single value of Y that

¹<http://cplint.eu/e/arithm.pl>

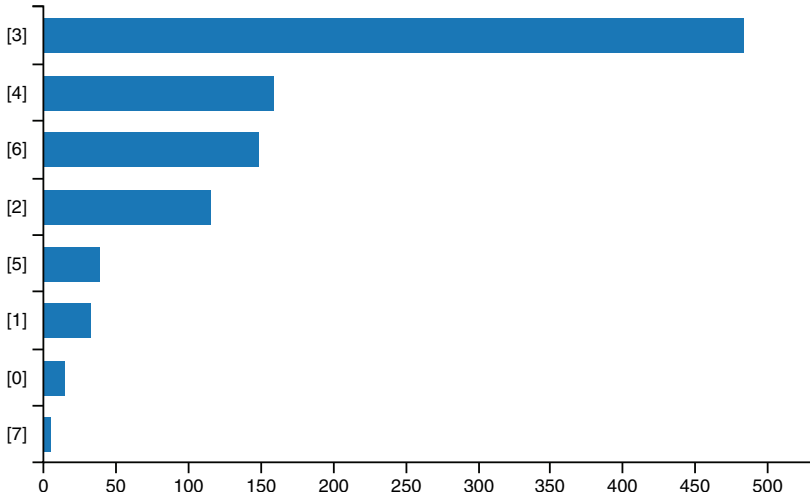


Figure 7.6 Distribution of sampled values in the Program of Example 90.

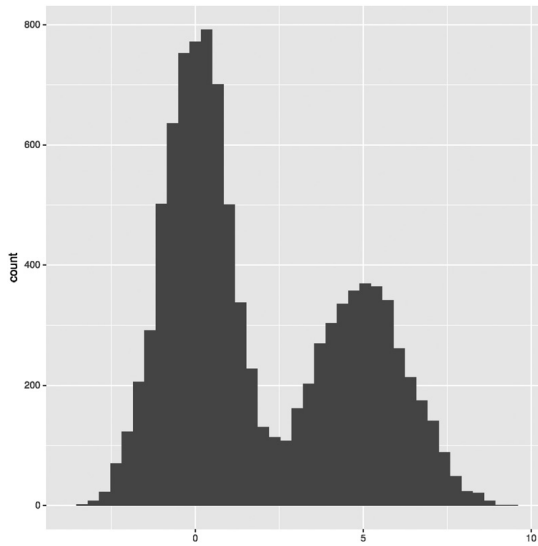


Figure 7.7 Distribution of sampled values from the Gaussian mixture of Example 91.

makes `eval(2, Y)` true in each world and the list of values in each sampled world contains a single element.

Example 91 (Gaussian mixture – sampling arguments – `cplint`). *Example 57 encodes of a mixture of two Gaussians with the program² that we report below*

```
heads:0.6;tails:0.4.  
g(X): gaussian(X, 0, 1).  
h(X): gaussian(X, 5, 2).  
mix(X) :- heads, g(X).  
mix(X) :- tails, h(X).
```

If we take 10000 samples of argument X of `mix(X)`, we may get the distribution of values that are shown in Figure 7.7.

²http://cplint.eu/e/gauss_mean_est.pl

8

Non-Standard Inference

This chapter discusses inference problems for languages that are related to PLP, such as Possibilistic Logic Programming, or are generalizations of PLP, such as Algebraic ProbLog. Moreover, the chapter illustrates how decision-theoretic problems can be solved by exploiting PLP techniques.

8.1 Possibilistic Logic Programming

Possibilistic Logic [Dubois et al., 1994] is a logic of uncertainty for reasoning under incomplete evidence. In this logic, the degree of *necessity* of a formula expresses to what extent the available evidence entails the truth of the formula and the degree of *possibility* expresses to what extent the truth of the formula is not incompatible with the available evidence.

Given a formula ϕ , we indicate with $\Pi(\phi)$ the degree of possibility assigned by *possibility measure* Π to it, and with $N(\phi)$ the degree of necessity assigned by *necessity measure* N to it. Possibility and necessity measures must satisfy the constraint $N(\phi) = 1 - \Pi(\neg\phi)$ for all formulas ϕ .

A *possibilistic clause* is a first-order logic clause C associated with a number that is a lower bound on its necessity or possibility degree. We consider here the possibilistic logic CPL1 [Dubois et al., 1991] in which only lower bounds on necessity are considered. Thus, (C, α) means that $N(C) \geq \alpha$. A *possibilistic theory* is a set of possibilistic clauses.

A possibility measure *satisfies a possibilistic clause* (C, α) if $N(C) \geq \alpha$ or, equivalently, if $\Pi(\neg C) \leq 1 - \alpha$. A possibility measure *satisfies a possibilistic theory* if it satisfies every clause in it. A possibilistic clause (C, α) is a *consequence* of a possibilistic theory F if every possibility measure satisfying F also satisfies (C, α) .

Inference rules of classical logic have been extended to rules in possibilistic logic. Here we report two sound inference rules [Dubois and Prade, 2004]:

- $(\phi, \alpha), (\psi, \beta) \vdash (R(\phi, \psi), \min(\alpha, \beta))$ where $R(\phi, \psi)$ is the resolvent of ϕ and ψ (extension of resolution);
- $(\phi, \alpha), (\phi, \beta) \vdash (\phi, \max(\alpha, \beta))$ (weight fusion).

Dubois et al. [1991] proposed a Possibilistic Logic Programming language. A program in such a language is a set of formulas of the form (C, α) where C is a definite clause

$$h \leftarrow b_1, \dots, b_n.$$

and α is a possibility or necessity degree. We consider the subset of this language that is included in CPL1, i.e., α is a real number in $(0,1]$ that is a lower bound on the necessity degree of C . The problem of inference in this language consists in computing the maximum value of α such that $N(q) \geq \alpha$ holds for a query q . The above inference rules are complete for this language.

Example 92 (Possibilistic logic program). *The following possibilistic program computes the least unsure path in a graph, i.e., the path with maximal weight, the weight of a path being the weight of its weakest edge [Dubois et al., 1991].*

$$\begin{array}{ll} (\text{path}(X, X), & 1) \\ (\text{path}(X, Y) \leftarrow \text{path}(X, Z), \text{edge}(Z, Y), & 1) \\ (\text{edge}(a, b), & 0.3) \\ \dots & \end{array}$$

We restrict our discussion here to positive programs. However, approaches for normal Possibilistic Logic programs have been proposed in [Nieves et al., 2007; Nicolas et al., 2006; Osorio and Nieves, 2009], and [Bauters et al., 2010].

PITA(IND,IND), see Section 5.9, can also be used to perform inference in Possibilistic Logic Programming where a program is composed only of clauses of the form $h : \alpha \leftarrow b_1, \dots, b_n$ which are interpreted as possibilistic clauses of the form $(h \leftarrow b_1, \dots, b_n, \alpha)$.

The PITA transformation of PITA(IND,IND) can be used unchanged provided that the support predicates are defined as

$equality([P, \neg P0], \neg N, P).$
 $or(A, B, C) \leftarrow C \text{ is } max(A, B).$
 $and(A, B, C) \leftarrow C \text{ is } min(A, B).$
 $zero(0.0).$
 $one(1.0).$
 $ret_prob(P, P).$

We obtain in this way PITA(POSS). The input list of the $equality/3$ predicate contains two numbers because we use the PITA transformation unchanged. Specializing it for possibilistic logic programs would remove the need for the $equality/3$ predicate.

Computing the possibility is much easier than computing the general probability, which must solve the disjoint sum problem to obtain answers.

8.2 Decision-Theoretic ProbLog

Decision-Theoretic ProbLog [Van den Broeck et al., 2010] or DTPROBLOG tackles *decision problems*: the selection of actions from a set of alternatives so that a utility function is maximized. In other words, the problem is to choose the actions that bring the most expected reward (or the least expected cost) to the acting agent. DTPROBLOG supports decision problems where the domain is described using ProbLog so that probabilistic effects of actions can be taken into account.

DTPROBLOG extends ProbLog by adding decision facts \mathcal{D} and utility facts \mathcal{U} . *Decision facts* model decision variables, i.e., variables on which we can act by setting their truth value. They are represented as

$$? :: d.$$

where d is an atom, possibly non-ground.

A *utility fact* is of the form

$$u \rightarrow r$$

where u is a literal and $r \in \mathbb{R}$ is a reward or utility for achieving u . It may be interpreted as a query that, when succeeding, gives a reward of r . u may be non-ground; in this case, the reward is given once if any grounding succeeds.

A *strategy* σ is a function $\mathcal{D} \rightarrow [0, 1]$ that assigns a decision fact to a probability. All groundings of the same decision fact are assigned the same probability. We call Σ the set of all possible strategies. We indicate with $\sigma(\mathcal{D})$ the set of probabilistic facts obtained by assigning probability $\sigma(d)$ to each decision fact $? :: d$. of \mathcal{D} , i.e., $\sigma(\mathcal{D}) = \{\sigma(d) :: d|? :: d \in \mathcal{D}\}$. A deterministic strategy is a strategy that only assigns probabilities 0 and 1 to decision facts. It is thus equivalent to a Boolean assignments to the decision atoms.

Given a DTPROBLOG $\mathcal{DT} = \mathcal{BK} \cup \mathcal{D}$ program and a strategy σ , the probability to a query q is the probability $P_\sigma(q)$ assigned by the ProbLog program $\mathcal{BK} \cup \sigma(\mathcal{D})$.

The *utility* of a logic program P given a set of utility facts \mathcal{U} is defined as

$$\text{Util}(P) = \sum_{u \rightarrow r \in \mathcal{U}, P \models u} r$$

The *expected utility* of a ProbLog program \mathcal{P} given a set of utility facts \mathcal{U} can thus be defined as

$$\text{Util}(\mathcal{P}) = \sum_{w \in W_{\mathcal{P}}} P(w) \sum_{u \rightarrow r \in \mathcal{U}, w \models u} r.$$

By exchanging the sum, we get

$$\text{Util}(\mathcal{P}) = \sum_{u \rightarrow r \in \mathcal{U}} \sum_{w \in W_{\mathcal{P}}, w \models u} r \cdot P(w) = \sum_{u \rightarrow r \in \mathcal{U}} r P(u)$$

The *expected utility* of a DTPROBLOG program $\mathcal{DT} = \mathcal{BK} \cup \mathcal{D} \cup \mathcal{U}$ and a strategy σ is the expected utility of $\mathcal{BK} \cup \sigma(\mathcal{D})$ given \mathcal{U}

$$\text{Util}(\sigma(\mathcal{DT})) = \text{Util}(\mathcal{BK} \cup \sigma(\mathcal{D})).$$

If we call $\text{Util}(u, \sigma(\mathcal{DT})) = r \cdot P(u)$ the expected utility due to atom u for a strategy σ , we have

$$\text{Util}(\sigma(\mathcal{DT})) = \sum_{u \rightarrow r \in \mathcal{U}} \text{Util}(u, \sigma(\mathcal{DT})).$$

Example 93 (Remaining dry [Van den Broeck et al., 2010]). *Consider the problem of remaining dry even when the weather is unpredictable. The possible actions are wearing a raincoat and carrying an umbrella:*

$? :: \text{umbrella.}$
 $? :: \text{raincoat.}$
 $0.3 :: \text{rainy.}$
 $0.5 :: \text{windy.}$
 $\text{broken_umbrella} \leftarrow \text{umbrella, rainy, windy.}$
 $\text{dry} \leftarrow \text{rainy, umbrella, } \sim\text{broken_umbrella.}$
 $\text{dry} \leftarrow \text{rainy, raincoat.}$
 $\text{dry} \leftarrow \sim\text{rainy.}$
Utility facts associate real numbers to atoms
 $\text{umbrella} \rightarrow -2 \quad \text{dry} \rightarrow 60$
 $\text{raincoat} \rightarrow -20 \quad \text{broken_umbrella} \rightarrow -40$

The *inference problem* in DTPROBLOG is to compute $\text{Util}(\sigma(\mathcal{DT}))$ for a particular strategy σ .

The *decision problem* instead consists of finding the optimal strategy, i.e., the one that provides the maximum expected utility. Formally, it means solving

$$\arg \max_{\sigma} \text{Util}(\sigma(\mathcal{DT})).$$

Since all the decisions are independent, we can consider only deterministic strategies. In fact, if the derivative of the total utility with respect to the probability assigned to a decision variable is positive (negative), the best result is obtained by assigning probability 1 (0). If the derivative is 0, it does not matter.

The inference problem is solved in DTPROBLOG by computing $P(u)$ with probabilistic inference for all decision facts $u \rightarrow r$ in \mathcal{U} . DTPROBLOG uses compilation of the query to BDDs as ProbLog1.

Example 94 (Continuation of Example 93). *For the utility fact dry, ProbLog1 builds the BDD of Figure 8.1. For the strategy*

$$\sigma = \{\text{umbrella} \rightarrow 1, \text{raincoat} \rightarrow 0\},$$

the probability of dry is $0.7 + 0.3 \cdot 0.5 = 0.85$, so $\text{Util}(\text{dry}, \sigma(\mathcal{DT})) = 60 \cdot 0.85 = 51$.

For the utility fact broken_umbrella, ProbLog1 builds the BDD of Figure 8.2. For the strategy $\{\text{umbrella} \rightarrow 1, \text{raincoat} \rightarrow 0\}$, the probability of broken_umbrella is $0.3 \cdot 0.5 = 0.15$ and $\text{Util}(\text{broken_umbrella}, \sigma(\mathcal{DT})) = -40 \cdot 0.15 = -6$.

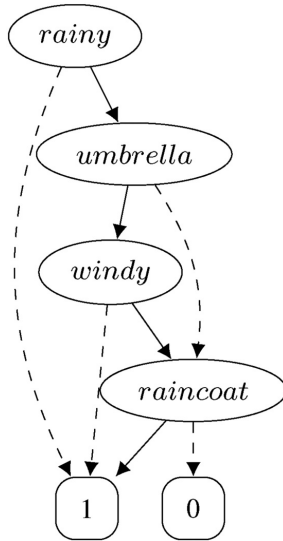


Figure 8.1 BDD_{dry}(σ) for Example 93. From [Van den Broeck et al., 2010].

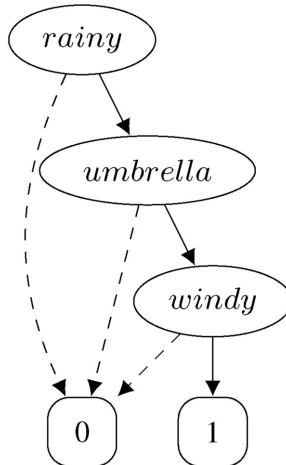


Figure 8.2 BDD_{broken_umbrella}(σ) for Example 93. From [Van den Broeck et al., 2010].

Overall, we get

$$\text{Util}(\sigma(\mathcal{DT})) = 51 + (-6) + (-2) = 43.$$

To solve decision problems, DTPROBLOG uses Algebraic Decision Diagrams (ADDs) [Bahar et al., 1997] that are a generalization of BDDs where

the leaves store a value in \mathbb{R} instead of 0 or 1. An ADD thus represents a function from Boolean variables to the real numbers $f : \{0, 1\}^n \rightarrow \mathbb{R}$ using a form of Shannon expansion:

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + (1 - x_1) \cdot f(0, x_2, \dots, x_n).$$

As BDDs, ADDs can be combined with operations. We consider here scalar multiplication $c \cdot g$ of an ADD g with the constant c , addition $f \oplus g$ of two ADDs, and if-then-else $\text{ITE}(b, f, g)$ where b is a Boolean variable.

In scalar multiplication, $h = c \cdot g$ with $c \in \mathbb{R}$ and $g : \{0, 1\}^n \rightarrow \mathbb{R}$, the output $h : \{0, 1\}^n \rightarrow \mathbb{R}$ is defined as: $\forall \mathbf{x} : h(\mathbf{x}) = c \cdot g(\mathbf{x})$.

In addition, $h = f \oplus g$ with $f, g : \{0, 1\}^n \rightarrow \mathbb{R}$, the output $h : \{0, 1\}^n \rightarrow \mathbb{R}$ is defined as $\forall \mathbf{x} : h(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x})$.

The version of if-then-else of interest here, $\text{ITE}(b, f, g)$ with $b \in \{0, 1\}$ and $f, g : \{0, 1\}^n \rightarrow \mathbb{R}$, returns $h : \{0, 1\}^{n+1} \rightarrow \mathbb{R}$ computed as

$$\forall b, \mathbf{x} : h(b, \mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{if } b = 1 \\ g(\mathbf{x}) & \text{if } b = 0 \end{cases}.$$

The CUDD [Somenzi, 2015] package, for example, offers these operations.

DTPROBLOG stores three functions:

- $P_\sigma(u)$, the probability of literal u as a function of the strategy;
- $\text{Util}(u, \sigma(\mathcal{DT}))$, the expected utility of literal u as a function of the strategy;
- $\text{Util}(\sigma)$, the total expected utility as a function of the strategy.

Since we can consider only deterministic strategies, one of them can be represented with a Boolean vector \mathbf{d} with an entry for each decision variable d_i . Therefore, all these functions are Boolean functions and can be represented with the ADDs $\text{ADD}(u)$, $\text{ADD}^{\text{util}}(u)$, and $\text{ADD}_{\text{tot}}^{\text{util}}$, respectively.

Given $\text{ADD}_{\text{tot}}^{\text{util}}$, finding the best strategy is easy: we just have to identify the leaf with the highest value and return a path from the root to the leaf, represented as a Boolean vector \mathbf{d} .

To build $\text{ADD}(u)$, DTPROBLOG builds the BDD $\text{BDD}(u)$ representing the truth of the query u as a function of the probabilistic and decision facts: given an assignment \mathbf{f}, \mathbf{d} for those, $\text{BDD}(u)$ returns either 0 or 1. $\text{BDD}(u)$ represents the probability $P(u|\mathbf{f}, \mathbf{d})$ for all values of \mathbf{f}, \mathbf{d} .

Function $P_\sigma(u)$ requires $P(u|\mathbf{d})$ that can be obtained from $P(u|\mathbf{f}, \mathbf{d})$ by summing out the \mathbf{f} variables, i.e., computing

$$\begin{aligned} P(u|\mathbf{d}) &= \sum_{\mathbf{f}} P(u, \mathbf{f}|\mathbf{d}) = \sum_{\mathbf{f}} P(u|\mathbf{f}, \mathbf{d})P(\mathbf{f}|\mathbf{d}) = \\ &= \sum_{\mathbf{f}} P(u|\mathbf{f}, \mathbf{d})P(\mathbf{f}) = \sum_{\mathbf{f}} P(u|\mathbf{f}, \mathbf{d}) \prod_{f \in \mathbf{f}} \Pi_f = \\ &= \sum_{f_1} \Pi_{f_1} \sum_{f_2} \Pi_{f_2} \dots \sum_{f_n} \Pi_{f_n} P(u|f_1, \dots, f_n, \mathbf{d}) \end{aligned}$$

We can obtain $\text{ADD}(u)$ from $\text{BDD}(u)$ by traversing $\text{BDD}(u)$ from the leaves to the root and, for each sub-BDD with root in node n , building a corresponding sub-ADD with root node m . If n is the 0-terminal (1-terminal), we return a 0-terminal (1-terminal). If n is associated with a probabilistic variable f , we have already built the ADDs ADD_l and ADD_h for its 0- and 1-child. They represent $P(u|f = 0, \mathbf{d}')$ and $P(u|f = 1, \mathbf{d}')$, respectively, for all values \mathbf{d}' where \mathbf{D}' is the set of Boolean decision variables of ADD_l and ADD_h . We must sum out variable f , so

$$P(u|\mathbf{d}') = \Pi_f \cdot P(u|f = 0, \mathbf{d}') + (1 - \Pi_f) \cdot P(u|f = 1, \mathbf{d}')$$

The ADD

$$\Pi_f \cdot \text{ADD}_l \oplus (1 - \Pi_f) \cdot \text{ADD}_h$$

represents $P(u|\mathbf{d}')$ and is the ADD we are looking for.

If n is associated with decision variable d with ADDs ADD_l and ADD_h for its 0- and 1-child, the ADD for representing $P(u|d, \mathbf{d}')$ is

$$\text{ITE}(d, \text{ADD}_h, \text{ADD}_l).$$

The conversion from $\text{BDD}(u)$ to $\text{ADD}(u)$ is computed by function `PROBABILITYDD` of Algorithm 11.

Once we have $\text{ADD}(u)$, $\text{ADD}^{util}(u)$ is simply given by $r \cdot \text{ADD}(u)$ if $u \rightarrow r \in \mathcal{U}$. Finally, $\text{ADD}_{tot}^{util} = \bigoplus_{u \rightarrow r \in \mathcal{U}} \text{ADD}^{util}(u)$. This gives Algorithm 11 that solves the decision problem exactly. The function `EXACT-SOLUTION` initializes ADD_{tot}^{util} to the zero function and then cycles over each utility fact in turn, building $\text{BDD}(u)$, $\text{ADD}(u)$, and $\text{ADD}^{util}(u)$. Then ADD_{tot}^{util} is updated by summing $\text{ADD}^{util}(u)$ to the current value.

Example 95 (Continuation of Example 93). *For the utility fact `dry`, `DTPROBLOG` builds $\text{ADD}(\text{dry})$ and $\text{ADD}^{util}(\text{dry})$ of Figure 8.3. For the strategy*

$$\sigma = \{\text{umbrella} \rightarrow 1, \text{raincoat} \rightarrow 0\},$$

Algorithm 11 Function EXACTSOLUTION: Solving the DTPROBLOG decision problem exactly.

```

1: function EXACTSOLUTION( $\mathcal{DT}$ )
2:    $ADD_{tot}^{util} \leftarrow 0$ 
3:   for all  $(u \rightarrow r) \in \mathcal{U}$  do
4:     Build  $BDD(u)$ , the BDD for  $u$ 
5:      $ADD(u) \leftarrow \text{PROBABILITYDD}(BDD_u(\mathcal{DT}))$ 
6:      $ADD^{util}(u) \leftarrow r \cdot ADD_u(\sigma)$ 
7:      $ADD_{tot}^{util} \leftarrow ADD_{tot}^{util} \oplus ADD^{util}(u)$ 
8:   end for
9:   let  $t_{max}$  be the terminal node of  $ADD_{tot}^{util}$  with the highest utility
10:  let  $p$  be a path from  $t_{max}$  to the root of  $ADD_{tot}^{util}$ 
11:  return the Boolean decisions made on  $p$ 
12: end function
13: function PROBABILITYDD( $n$ )
14:  if  $n$  is the 1-terminal then
15:    return a 1-terminal
16:  end if
17:  if  $n$  is the 0-terminal then
18:    return a 0-terminal
19:  end if
20:  let  $h$  and  $l$  be the high and low children of  $n$ 
21:   $ADD_h \leftarrow \text{PROBABILITYDD}(h)$ 
22:   $ADD_l \leftarrow \text{PROBABILITYDD}(l)$ 
23:  if  $n$  represents a decision  $d$  then
24:    return  $\text{ITE}(d, ADD_h, ADD_l)$ 
25:  end if
26:  if  $n$  represents a fact with probability  $p$  then
27:    return  $(p \cdot ADD_h) \oplus ((1 - p) \cdot ADD_l)$ 
28:  end if
29: end function

```

the figure confirms that $\text{Util}(\text{dry}, \sigma(\mathcal{DT})) = 60 \cdot 0.85 = 51$.

For *broken_umbrella*, DTPROBLOG builds $ADD(\text{broken_umbrella})$ and $ADD^{util}(\text{broken_umbrella})$ of Figure 8.4. For the strategy σ , the figure confirms that

$$\text{Util}(\text{broken_umbrella}, \sigma(\mathcal{DT})) = -40 \cdot 0.15 = -6.$$

Figure 8.5 shows ADD_{tot}^{util} that confirms that, for strategy σ ,

$$\text{Util}(\sigma(\mathcal{DT})) = 43.$$

Moreover, this is also the optimal strategy.

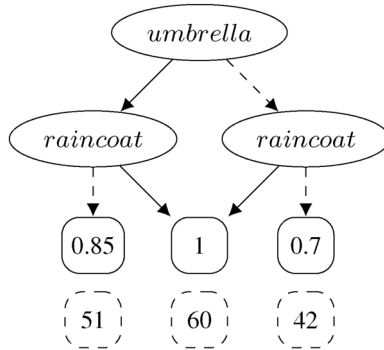


Figure 8.3 $ADD(dry)$ for Example 93. The dashed terminals indicate $ADD^{util}(dry)$. From [Van den Broeck et al., 2010].

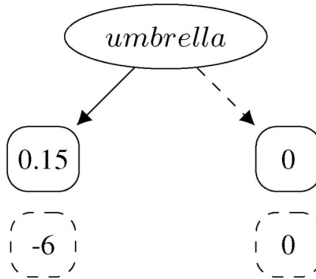


Figure 8.4 $ADD(broken_umbrella)$ for Example 93. The dashed terminals indicate $ADD^{util}(broken_umbrella)$. From [Van den Broeck et al., 2010].

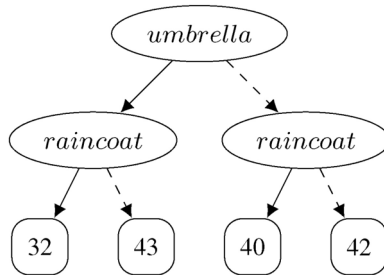


Figure 8.5 ADD_{tot}^{util} for Example 93. From [Van den Broeck et al., 2010].

Algorithm 11 can be optimized by pruning ADD_{tot}^{util} to remove the portions that may never lead to an optimal strategy. Denote by $\max ADD$ and $\min ADD$ the maximum and minimum value, respectively, that ADD assigns to any combination of its variables, i.e., the maximum and minimum values that appear in its leaves.

When $\text{ADD}^{util}(u_i)$ is summed to ADD_{tot}^{util} , a leaf of ADD_{tot}^{util} with value v before the sum belongs to $[v + \min \text{ADD}_{tot}^{util}(u_i), v + \max \text{ADD}_{tot}^{util}(u_i)]$ after the sum. Thus, if m is $\max \text{ADD}_{tot}^{util}$ before the sum, any leaf with value v such that $v + \max \text{ADD}_{tot}^{util}(u_i) < m + \min \text{ADD}_{tot}^{util}(u_i)$ will never lead to an optimal strategy. All such leaves can be pruned by merging them and assigning them the value $-\infty$.

If we compute the impact of utility attribute u_i as

$$\text{Im}(u_i) = \max \text{ADD}_{tot}^{util}(u_i) - \min \text{ADD}_{tot}^{util}(u_i)$$

we can prune all the leaves of ADD_{tot}^{util} that, before the sum, have a value below

$$\max \text{ADD}_{tot}^{util} - \text{Im}(u_i)$$

We then perform the summation with the simplified ADD_{tot}^{util} which is cheaper than the summation with the original ADD.

Moreover, we can also consider the utility attributes still to be added and prune all the leaves of ADD_{tot}^{util} that have a value below

$$\max \text{ADD}_{tot}^{util} - \sum_{j \geq i} \text{Im}(u_j)$$

Finally, we can sort the utility attributes in order of decreasing $\text{Im}(u_i)$ and add them in this order, so that the maximum pruning is achieved.

The decision problem can also be solved approximately by adopting two techniques that can be used individually or in combination.

The first technique uses local search: a random strategy σ is initially selected by randomly assigning values to the decision variables and then a cycle is entered in which a single decision is flipped obtaining strategy σ' . If $\text{Util}(\sigma'(\mathcal{DT}))$ is larger than $\text{Util}(\sigma(\mathcal{DT}))$, the modification is retained and σ' becomes the current best strategy. $\text{Util}(\sigma(\mathcal{DT}))$ can be computed using the BDD $\text{BDD}(u)$ for each utility attribute using function **PROB** of Algorithm 4 for computing $P(u)$.

The second technique involves computing an approximation of the utility by using k -best (see Section 7.1.2) for building the BDDs for utility attributes.

Example 96 (Viral marketing [Van den Broeck et al., 2010]). *A firm is interested in marketing a new product to its customers. These are connected in a social network that is known to the firm: the network represents the trust relationships between customers. The firm wants to choose the customers on which to perform marketing actions. Each action has a cost, and a reward is given for each person that buys the product.*

We can model this domain with the DTPROBLOG program

```
? :: market(P) :- person(P).
0.4 :: viral(P,Q).
0.3 :: from_marketing(P).
market(P) -> -2 :- person(P).
buys(P) -> 5 :- person(P).
buys(P) :- market(P), from_marketing(P).
buys(P) :- trusts(P,Q), buys(Q), viral(P,Q).
```

Here the notation $\text{:- person}(P)$ after decision and utility facts means that there is a fact for each grounding of $\text{person}(P)$, i.e., a fact for each person. The program states that a person P buys the product if he is the target of a marketing action and the action causes the person to buy the product ($\text{from_marketing}(P)$), or if he trusts a person Q that buys the product and there is a viral effect ($\text{viral}(P, Q)$). The decisions are $\text{market}(P)$ for each person P . A reward of 5 is assigned for each person that buys the product and each marketing action costs 2.

Solving the decision problem for this program means deciding on which person to perform a marketing action so that the expected utility is maximized.

8.3 Algebraic ProbLog

Algebraic ProbLog (aProbLog) [Kimmig, 2010; Kimmig et al., 2011b] generalizes ProbLog to deal with labels of facts that are more general than probabilities. In particular, the labels are required to belong to a *semiring*, an algebraic structure.

Definition 44 (Semiring). A semiring is a tuple $(\mathcal{A}, \oplus, \otimes, e^\oplus, e^\otimes)$ such that

- \mathcal{A} is a set;
- \oplus is a binary operations over \mathcal{A} called addition that is commutative and associative and has neutral element e^\oplus , i.e., $\forall a, b, c \in \mathcal{A}$,

$$\begin{aligned} a \oplus b &= b \oplus a \\ (a \oplus b) \oplus c &= a \oplus (b \oplus c) \\ a \oplus e^\oplus &= a. \end{aligned}$$

- \otimes is a binary operations over \mathcal{A} called multiplication that left and right distributes over addition and has neutral element e^\otimes , i.e., $\forall a, b, c \in \mathcal{A}$,

$$\begin{aligned}(a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c) \\ a \otimes (b \oplus c) &= (a \otimes b) \oplus (a \otimes c) \\ a \otimes e^\otimes &= e^\otimes \otimes a = a.\end{aligned}$$

- e^\oplus annihilates \mathcal{A} , i.e., $\forall a \in \mathcal{A}$

$$a \otimes e^\oplus = e^\oplus \otimes a = e^\oplus.$$

A commutative semiring is a semiring $(\mathcal{A}, \oplus, \otimes, e^\oplus, e^\otimes)$ such that multiplication is commutative, i.e., $\forall a, b \in \mathcal{A}$,

$$a \otimes b = b \otimes a$$

An example of a commutative semiring is $([0, 1], +, \times, 0, 1)$ where $[0, 1] \subseteq \mathbb{R}$ and $+$ and \times are addition and multiplication over the reals. ProbLog associates each probabilistic fact with an element of $[0, 1]$ and computes the probability of queries by using addition and multiplication, so it can be seen as operating on the semiring $([0, 1], +, \times, 0, 1)$ that we call *probabilistic semiring*.

Another example of a commutative semiring is $([0, 1], \max, \min, 0, 1)$ where \max and \min are maximum and minimum operations over the reals. PITA(POSS) can be seen as operating on the semiring $([0, 1], \max, \min, 0, 1)$ that we call *possibilistic semiring*.

ProbLog can be generalized to operate on a semiring. For a set of ground atoms A , let $L(A)$ be the set of literals that can be built on atoms from A , i.e., $L(A) = A \cup \{\sim a \mid a \in A\}$. For a two-valued interpretation J over a set of ground atoms A , define the *complete interpretation* $c(J) = J \cup \{\sim a \mid a \in A \setminus J\}$ and the set of all possible complete interpretations $\mathcal{I}(A) = \{c(J) \mid J \subseteq A\}$. Consistent sets of literals built on atoms A form the set $\mathcal{C}(A) = \{H \mid H \subseteq I, I \in \mathcal{I}(A)\}$.

Definition 45 (aProbLog [Kimmig et al., 2011b]). *An aProbLog program consists of*

- a commutative semiring $(\mathcal{A}, \oplus, \otimes, e^\oplus, e^\otimes)$;
- a finite set of ground atoms $\mathcal{F} = \{f_1, \dots, f_n\}$ called algebraic facts;

- a finite set of rules \mathcal{R} ;
- a labeling function $\alpha : L(\mathcal{F}) \rightarrow \mathcal{A}$.

The possible complete interpretations for \mathcal{F} are also called *worlds* and $\mathcal{I}(\mathcal{F})$ is the set of all worlds. aProbLog assigns labels to worlds and sets of worlds as follows. The label of a world $I \in \mathcal{I}(\mathcal{F})$ is the product of the labels of its literals

$$\mathbf{A}(I) = \bigotimes_{l \in I} \alpha(l)$$

The label of a set of complete interpretations $S \subseteq \mathcal{I}(\mathcal{F})$ is the sum of the labels of each interpretation

$$\mathbf{A}(S) = \bigoplus_{I \in S} \bigotimes_{l \in I} \alpha(l).$$

A query is a set of ground literals. Given a query q , we denote the set of complete interpretations where the query is true as $\mathcal{I}(q)$ defined as

$$\mathcal{I}(q) = \{I \mid I \in \mathcal{I}(\mathcal{F}) \wedge I \cup \mathcal{R} \models q\}$$

The label of the query q is then defined as the label of $\mathcal{I}(q)$:

$$\mathbf{A}(q) = \mathbf{A}(\mathcal{I}(q)) = \bigoplus_{I \in \mathcal{I}(q)} \bigotimes_{l \in I} \alpha(l).$$

Since both operations are commutative and associative, the label of a query does not depend on the order of literals and of interpretations.

The inference problem in aProbLog consists of computing the labels of queries. Depending on the choice of commutative semiring, an inference task may correspond to a certain known problem or new problems. Table 8.1 lists some known inference tasks with their corresponding commutative semirings.

Table 8.1 Inference tasks and corresponding semirings for aProbLog. Adapted from [Kimmig et al., 2011b]

task	\mathcal{A}	e^\oplus	e^\otimes	$a \oplus b$	$a \otimes b$	$\alpha(f)$	$\alpha(\sim f)$
PROB	$[0, 1]$	0	1	$a + b$	$a \cdot b$	$\alpha(f)$	$1 - \alpha(f)$
POSS	$[0, 1]$	0	1	$\max(a, b)$	$\min(a, b)$	$\alpha(f)$	1
MPE	$[0, 1]$	0	1	$\max(a, b)$	$a \cdot b$	$\alpha(f)$	$1 - \alpha(f)$
MPE State	$[0, 1] \times \mathbb{P}(\mathcal{C}(\mathcal{F}))$	$(0, \emptyset)$	$(1, \{\emptyset\})$	Eq. 8.2	Eq. 8.1	$(p, \{\{f\}\})$	$(1 - p, \{\{\sim f\}\})$
SAT	$\{0, 1\}$	0	1	$a \vee b$	$a \wedge b$	1	1
#SAT	\mathbb{N}	0	1	$a + b$	$a \cdot b$	1	1
BDD	$\text{BDD}(\mathcal{V})$	$bdd(0)$	$bdd(1)$	$a \vee b \text{ d d } b$	$a \wedge b \text{ d d } b$	$bdd(f)$	$\neg b \text{ d d } b \text{ d d } (f)$
Sensitivity	$\mathbb{R}[\mathbf{X}]$	0	1	$a + b$	$a \cdot b$	x or in $[0, 1]$	$1 - \alpha(f)$
Gradient	$[0, 1] \times \mathbb{R}$	$(0, 0)$	$(1, 0)$	Eq. 8.3	Eq. 8.4	Eq. 8.5	Eq. 8.6

Example 97 (Alarm – aProbLog [Kimmig et al., 2011b]). Consider the following aProbLog program similar to the ProbLog program of Example 69:

$calls(X) \leftarrow alarm, hears_alarm(X).$

$alarm \leftarrow burglary.$

$alarm \leftarrow earthquake.$

$0.7 :: hears_alarm(john).$

$0.7 :: hears_alarm(mary).$

$0.05 :: burglary.$

$0.01 :: earthquake.$

where the labels of positive literals are attached to each fact f . This program is a variation of the alarm BN of Example 10. The program has 16 worlds and the query $calls(mary)$ is true in six of them, those shown in Figure 8.6. For the PROB task, we consider the probability semiring and the label of negative literals are defined as $\alpha(\sim f) = 1 - \alpha(f)$. Then the label of $calls(mary)$ is

$$\begin{aligned} \mathbf{A}(calls(mary)) &= 0.7 \cdot 0.7 \cdot 0.05 \cdot 0.01 + \\ & 0.7 \cdot 0.7 \cdot 0.05 \cdot 0.99 + \\ & 0.7 \cdot 0.7 \cdot 0.95 \cdot 0.01 + \\ & 0.3 \cdot 0.7 \cdot 0.05 \cdot 0.01 + \\ & 0.3 \cdot 0.7 \cdot 0.05 \cdot 0.99 + \\ & 0.3 \cdot 0.7 \cdot 0.95 \cdot 0.01 = \\ & 0.04165 \end{aligned}$$

For the MPE task, the semiring is $([0, 1], +, \max, 0, 1)$, the label of negative literals are defined as $\alpha(\sim f) = 1 - \alpha(f)$, and the label of $calls(mary)$ is

$$\begin{aligned} \mathbf{A}(calls(mary)) &= 0.7 \cdot 0.7 \cdot 0.05 \cdot 0.01 + \\ & 0.7 \cdot 0.7 \cdot 0.05 \cdot 0.99 = \\ & 0.001995 \end{aligned}$$

{	$hears_alarm(john),$	$hears_alarm(mary),$	$burglary,$	$earthquake$	}
{	$hears_alarm(john),$	$hears_alarm(mary),$	$burglary,$	$\sim earthquake$	}
{	$hears_alarm(john),$	$hears_alarm(mary),$	$\sim burglary,$	$earthquake$	}
{	$\sim hears_alarm(john),$	$hears_alarm(mary),$	$burglary,$	$earthquake$	}
{	$\sim hears_alarm(john),$	$hears_alarm(mary),$	$burglary,$	$\sim earthquake$	}
{	$\sim hears_alarm(john),$	$hears_alarm(mary),$	$\sim burglary,$	$earthquake$	}

Figure 8.6 Worlds where the query $calls(mary)$ from Example 97 is true.

For the SAT task, the semiring is $(\{0, 1\}, \vee, \wedge, 0, 1)$, the label of literals is always 1, and the label of $calls(mary)$ is 1 as there are six worlds where the query is true.

The MPE State task is an extension of the MPE task that also returns the world with the highest label. The set \mathcal{A} is $[0, 1] \times \mathbb{P}(\mathcal{CF})$ where $\mathbb{P}(\mathcal{CF})$ is the powerset of $\mathcal{C}(\mathcal{F})$, so the second element of the labels is a set of consistent sets of literals built on algebraic facts. The aim is for the label of queries to have as first argument the maximum of the probability of the worlds and as second argument the set of worlds with that probability (there can be more than one if they share the same probability). The operations are defined as

$$(p, S) \otimes (q, T) = (p \cdot q, \{I \cup J \mid I \in S, J \in T\}) \quad (8.1)$$

$$(p, S) \oplus (q, T) = \begin{cases} (p, S) & \text{if } p > q \\ (q, T) & \text{if } p < q \\ (p, S \cup T) & \text{if } p = q \end{cases} \quad (8.2)$$

The label for the query $mary(calls)$ of Example 97 is

$$\begin{aligned} \mathbf{A}(calls(mary)) &= (0.7 \cdot 0.7 \cdot 0.05 \cdot 0.99, I) = (0.001995, I) \\ I &= \{hears_alarm(john), hears_alarm(mary), burglary, \\ &\quad \sim earthquake\} \end{aligned}$$

We can count the number of satisfying assignment with the #SAT task with semiring $(\mathbb{N}, +, \times, 0, 1)$ and labels $\alpha(f_i) = \alpha(\sim f_i) = 1$. We can also have labels encoding functions or data structures. For example, labels may encode Boolean functions represented as BDDs and algebraic facts may be Boolean variables from a set \mathcal{V} . In this case, we can use the semiring

$$(\text{BDD}(\mathcal{V}), \vee_{bdd}, \wedge_{bdd}, bdd(0), bdd(1))$$

and assign labels as $\alpha(f_i) = bdd(f_i)$ and $\alpha(\sim f_i) = \neg_{bdd} bdd(f_i)$ where $\text{BDD}(\mathcal{V})$ is the set of BDDs over variables \mathcal{V} , $\vee_{bdd}, \wedge_{bdd}, \neg_{bdd}$ are Boolean operations over BDDs, and $bdd(\cdot)$ can be applied to the values 0, 1, $f \in F$ returning the BDD representing the 0, 1, or f Boolean functions.

aProbLog can also be used for sensitivity analysis, i.e., estimating how a change in the probability of the facts changes the probability of the query. In this case, the labels are polynomials over a set of variables (indicated with $\mathbb{R}[\mathbf{X}]$) in Table 8.1. In Example 97, if we use variables x and y to label the facts $burglary$ and $hears_alarm(mary)$, respectively, the label of $calls(mary)$ becomes $0.99 \cdot x \cdot y + 0.01 \cdot y$ that is also the probability of $calls(mary)$.

Another task is gradient computation where we want to compute the gradient of the probability of the query, as, for example, is done by LeProbLog, see Section 9.3. We consider here the case where we want to compute the derivative with respect to the parameter p_k of the k -th fact. The labels are pairs where the first element stores the probability and the second its derivative with respect to p_k . Using the rule for the derivative of a product, it is easy to see that the operations can be defined as

$$(a_1, a_2) \oplus (b_1, b_2) = (a_1 + b_1, a_2 + b_2) \quad (8.3)$$

$$(a_1, a_2) \otimes (b_1, b_2) = (a_1 \cdot b_1, a_1 \cdot b_2 + a_2 \cdot b_1) \quad (8.4)$$

and the labels of the algebraic facts as

$$\alpha(f_i) = \begin{cases} (p_i, 1) & \text{if } i = k \\ (p_i, 0) & \text{if } i \neq k \end{cases} \quad (8.5)$$

$$\alpha(\sim f_i) = \begin{cases} (1 - p_i, -1) & \text{if } i = k \\ (1 - p_i, 0) & \text{if } i \neq k \end{cases} \quad (8.6)$$

To perform inference, aProbLog avoids the generation of all possible worlds and computes a covering set of explanations for the query, as defined in Section 3.1, similarly to what ProbLog does for PROB. We represent here an explanation E as a set of literals built on \mathcal{F} that are sufficient for entailing the query, i.e., $\mathcal{R} \cup E \models q$, and a covering set of explanations $\mathcal{E}(q)$ as a set such that

$$\forall I \in \mathcal{I}(q), \exists J \in \mathcal{E}(q) : J \subseteq I$$

We define the label of an explanation E as

$$\mathbf{A}(E) = \mathbf{A}(\mathcal{I}(E)) = \bigoplus_{I \in \mathcal{I}(E)} \bigotimes_{l \in I} \alpha(l)$$

We call $\mathbf{A}(E)$ a *neutral sum* if

$$\mathbf{A}(E) = \bigoplus_{l \in E} \alpha(l)$$

If $\forall f \in F : \alpha(f) \oplus \alpha(\sim f) = e^\otimes$, then $\mathbf{A}(E)$ is a neutral sum.

We call $\bigoplus_{E \in \mathcal{E}(q)} \mathbf{A}(E)$ a *disjoint sum* if

$$\bigoplus_{E \in \mathcal{E}(q)} \mathbf{A}(E) = \bigoplus_{I \in \mathcal{I}(q)} \mathbf{A}(I)$$

\oplus is *idempotent* if $\forall a \in \mathcal{A} : a \oplus a = a$. If \oplus is idempotent, then $\bigoplus_{E \in \mathcal{E}(q)} \mathbf{A}(E)$ is a disjoint sum.

Given a covering set of explanations $\mathcal{E}(q)$, we define the *explanation sum* as

$$\mathbf{S}(\mathcal{E}(q)) = \bigoplus_{E \in \mathcal{E}(q)} \bigotimes_{l \in E} \alpha(l)$$

If $\mathbf{A}(E)$ is a neutral sum for all $E \in \mathcal{E}(q)$ and $\bigoplus_{E \in \mathcal{E}(q)} \mathbf{A}(E)$ is a disjoint sum, then the explanation sum is equal to the label of the query, i.e.,

$$\mathbf{S}(\mathcal{E}(q)) = \mathbf{A}(q).$$

In this case, inference can be performed by computing $\mathbf{S}(\mathcal{E}(q))$. Otherwise, the *neutral sum* and/or *disjoint sum problems* must be solved.

To tackle the neutral-sum problem, let $\text{free}(E)$ denote the variables not occurring in an explanations E :

$$\text{free}(E) = \{f \mid f \in \mathcal{F}, f \notin E, \sim f \notin E\}$$

We can thus express $\mathbf{A}(E)$ as

$$\mathbf{A}(E) = \bigotimes_{l \in E} \alpha(l) \otimes \bigotimes_{l \in \text{free}(E)} (\alpha(l) \oplus \alpha(\sim l))$$

given the properties of commutative semirings.

The sum $\mathbf{A}(E_0) \oplus \mathbf{A}(E_1)$ of two explanations can be computed by exploiting the following property. Let $V_i = \{f \mid f \in E_i \vee \sim f \in E_i\}$ be the variables appearing in explanation E_i , then

$$\mathbf{A}(E_0) \oplus \mathbf{A}(E_1) = (\mathbf{P}_1(E_0) \oplus \mathbf{P}_0(E_1)) \otimes \bigotimes_{f \in \mathcal{F} \setminus (V_0 \cup V_1)} (\alpha(f) \oplus \alpha(\sim f)) \quad (8.7)$$

with

$$\mathbf{P}_j(E_i) = \bigotimes_{l \in E_i} \alpha(l) \otimes \bigotimes_{f \in V_j \setminus V_i} (\alpha(f) \oplus \alpha(\sim f))$$

So we can evaluate $\mathbf{A}(E_0) \oplus \mathbf{A}(E_1)$ by taking into account the set of variables on which the two explanations depend.

To solve the disjoint-sum problem, aProbLog builds a BDD representing the truth of the query as a function of the algebraic facts. If sums are neutral, aProbLog assigns a label to each node n as

$$\text{label}(1) = e^\otimes$$

$$\text{label}(0) = e^\oplus$$

$$\text{label}(n) = (\alpha(n) \otimes \text{label}(h)) \oplus (\alpha(\sim n) \otimes \text{label}(l))$$

Algorithm 12 Function LABEL: aProbLog inference algorithm.

```

1: function LABEL( $n$ )
2:   if  $Table(n) \neq null$  then
3:     return  $Table(n)$ 
4:   else
5:     if  $n$  is the 1-terminal then
6:       return  $(e^{\otimes}, \emptyset)$ 
7:     end if
8:     if  $n$  is the 0-terminal then
9:       return  $(e^{\oplus}, \emptyset)$ 
10:    end if
11:    let  $h$  and  $l$  be the high and low children of  $n$ 
12:     $(H, V_h) \leftarrow LABEL(h)$ 
13:     $(L, V_l) \leftarrow LABEL(l)$ 
14:     $\mathbf{P}_i(h) \leftarrow H \otimes \bigotimes_{x \in V_i \setminus V_h} (\alpha(x) \oplus \alpha(\sim x))$ 
15:     $\mathbf{P}_h(l) \leftarrow L \otimes \bigotimes_{x \in V_h \setminus V_l} (\alpha(x) \oplus \alpha(\sim x))$ 
16:     $label(n) \leftarrow (\alpha(n) \otimes \mathbf{P}_i(h)) \oplus (\alpha(\sim n) \otimes \mathbf{P}_h(l))$ 
17:     $Table(n) \leftarrow (label(n), \{n\} \cup V_h \cup V_l)$ 
18:    return  $Table(n)$ 
19:  end if
20: end function

```

where h and l denote the high and low child of n . In fact, a full Boolean decision tree expresses $\mathcal{E}(q)$ as an expression of the form

$$\bigvee_{l_1} l_1 \wedge \dots \wedge \bigvee_{l_n} l_n \wedge \mathbf{1}(\{l_1, \dots, l_n\} \in \mathcal{E}(q))$$

where l_i is a literal built on variable f_i and $\mathbf{1}(\{l_1, \dots, l_n\} \in \mathcal{E}(q))$ is 1 if $\{l_1, \dots, l_n\} \in \mathcal{E}(q)$ and 0 otherwise. By the properties of semirings,

$$\mathbf{A}(q) = \bigoplus_{l_1} l_1 \otimes \dots \otimes \bigoplus_{l_n} l_n \otimes \mathbf{e}(\{l_1, \dots, l_n\} \in \mathcal{E}(q)) \quad (8.8)$$

where $\mathbf{e}(\{l_1, \dots, l_n\} \in \mathcal{E}(q))$ is e^{\otimes} if $\{l_1, \dots, l_n\} \in \mathcal{E}(q)$ and e^{\oplus} otherwise. So given a full Boolean decision tree, the label of the query can be computed by traversing the tree bottom-up, as for the computation of the probability of the query with Algorithm 4.

BDDs are obtained from full Boolean decision trees by repeatedly merging isomorphic subgraphs and deleting nodes whose children are the same until no more operations are possible. The merging operation does not affect Equation (8.8), as it simply identifies identical sub-expressions. The deletion

operation deletes a node n when its high and low children are the same node s . In this case, the label of node n would be

$$\mathbf{label}(n) = (\alpha(n) \otimes \mathbf{label}(s)) \oplus (\alpha(\sim n) \otimes \mathbf{label}(s))$$

that is equal to $\mathbf{label}(n)$ if sums are neutral. If not, Algorithm 12 is used that uses Equation (8.7) to take into account deleted nodes. Function LABEL is called after initializing $Table(n)$ to *null* for all nodes n . $Table(n)$ stores intermediate results similarly to Algorithm 4 to keep the complexity linear in the number of nodes.

9

Parameter Learning

This chapter discusses the problem of learning the parameters of probabilistic logic programs with a given structure. We are given data, in the form of ground atoms or interpretations, and a probabilistic logic program, and we want to find the parameters of the program that assign maximum probability to the examples.

9.1 PRISM Parameter Learning

The PRISM system included a parameter learning algorithm since the original article [Sato, 1995]. The learning task considered there is given in the following definition.

Definition 46 (PRISM parameter learning problem). *Given a PRISM program \mathcal{P} and a set of examples $E = \{e_1, \dots, e_T\}$ which are ground atoms, find the parameters of msw fact so that the likelihood of the atoms $L = \prod_{t=1}^T P(e_t)$ is maximized. Equivalently, find the parameters of msw fact so that the Log Likelihood (LL) of the atoms $LL = \sum_{t=1}^T \log P(e_t)$ is maximized.*

Example 98 (Bloodtype – PRISM [Sato et al., 2017]). *The following program*

```
values (gene, [a, b, o]) .
bloodtype(P) :-
  genotype(X, Y),
  ( X=Y -> P=X
  ; X=o -> P=Y
  ; Y=o -> P=X
  ; P=ab
  ) .
genotype(X, Y) :- msw(gene, X), msw(gene, Y) .
```

encodes how a person's blood type is determined by his genotype, formed by a pair of two genes (a, b or o).

Learning in PRISM can be performed using predicate `learn/1` that takes a list of ground atoms (the examples) as argument, as in

```
?- learn([count(bloodtype(a), 40), count
         (bloodtype(b), 20),
         count(bloodtype(o), 30), count
         (bloodtype(ab), 10)]).
```

where `count(At, N)` denotes the repetition of atom `At` `N` times. After parameter learning, the parameters can be obtained with predicate `show_sw/0`, e.g.,

```
?- show_sw.
Switch gene: unfixed: a (0.292329558535712)
b (0.163020241540856)
o (0.544650199923432)
```

These values represents the probability distribution over the values a, b, and o of switch gene.

PRISM looks for the maximum likelihood parameters of the *msw* atoms. However, these are not observed in the dataset, which contains only derived atoms. Therefore, relative frequency cannot be used for computing the parameters and an algorithm for learning from incomplete data must be used. One such algorithm is Expectation Maximization (EM) [Dempster et al., 1977].

To perform EM, we can associate a random variable X_{ij} with values $D = \{x_{i1}, \dots, x_{in_i}\}$ to the ground switch name $i\theta_j$ of $msw(i, x)$ with domain D , with θ_j being a grounding substitution for i . Let $g(i)$ be the set of such substitutions:

$$g(i) = \{j | \theta_j \text{ is a grounding substitution for } i \text{ in } msw(i, x)\}.$$

The EM algorithm alternates between the two phases:

- Expectation: computes $\mathbf{E}[c_{ik}|e]$ for all examples e , switches $msw(i, x)$ and $k \in \{1, \dots, n_i\}$, where c_{ik} is the number of times a variable X_{ij} takes value x_{ik} with j in $g(i)$. $\mathbf{E}[c_{ik}|e]$ is given by $\sum_{j \in g(i)} P(X_{ij} = x|e)$.
- Maximization: computes Π_{ik} for all $msw(i, x)$ and $k = 1, \dots, n_i - 1$ as

$$\Pi_{ik} = \frac{\sum_{e \in E} \mathbf{E}[c_{ik}|e]}{\sum_{e \in E} \sum_{k=1}^{n_i} \mathbf{E}[c_{ik}|e]}.$$

So, for each e , X_{ij} s, and x_{ik} s, we compute $P(X_{ij} = x_{ik}|e)$, the expected value of X_{ij} given the example, with $k \in \{1, \dots, n_i\}$. These expected values are then aggregated and used to complete the dataset for computing the parameters by relative frequency. If c_{ik} is the number of times a variable X_{ij} takes value x_{ik} for any j , $\mathbf{E}[c_{ik}|e]$ is its expected value given example e . if $\mathbf{E}[c_{ik}]$ is its expected value given all the examples, then

$$\mathbf{E}[c_{ik}] = \sum_{t=1}^T \mathbf{E}[c_{ik}|e_t]$$

and

$$\Pi_{ik} = \frac{\mathbf{E}[c_{ik}]}{\sum_{k=1}^{n_i} \mathbf{E}[c_{ik}]}.$$

If the program satisfies the exclusive-or assumption, $P(X_{ij} = x_{ik}|e)$ can be computed as

$$P(X_{ij} = x_{ik}|e) = \frac{P(X_{ij} = x_{ik}, e)}{P(e)} = \frac{\sum_{\kappa \in K_e, msw(i, x_{ik})\theta_j \in e} P(\kappa)}{P(e)}$$

where K_e is the set of explanations of e and each explanation κ is a set of msw atoms of the form $msw(i, x_{ik})$. So we sum the probability of explanations that contain

$$msw(i, x_{ik})\theta_j$$

and divide by the probability of e , which is the sum of the probability of all explanations. This leads to the naive learning function of Algorithm 13 [Sato, 1995] that iterates the expectation and maximization steps until the LL converges.

This algorithm is naive because there can be exponential numbers of explanations for the examples, as in the case of the HMM of Example 65. As for inference, a more efficient dynamic programming algorithm can be devised that does not require the computation of all the explanations in case the program also satisfies the independent-and assumption [Sato and Kameya, 2001]. Tabling is used to find formulas of the form

$$g_i \Leftrightarrow S_{i1} \vee \dots \vee S_{is_i}$$

where the g_i s are subgoals in the derivation of an example e that can be ordered as $\{g_1, \dots, g_m\}$ such that $e = g_1$ and each S_{ij} contains only msw atoms and subgoals from $\{g_{i+1}, \dots, g_m\}$.

Algorithm 13 Function PRISM-EM: Naive EM learning in PRISM.

```

1: function PRISM-EM-NAIVE( $E, \mathcal{P}, \epsilon$ )
2:    $LL = -inf$ 
3:   repeat
4:      $LL_0 = LL$ 
5:     for all  $i, k$  do ▷ Expectation step
6:        $\mathbf{E}[c_{ik}] \leftarrow \sum_{e \in E} \frac{\sum_{\kappa \in K_e, ms w(i, x_{ik}) \theta_j \in e} P(\kappa)}{P(e)}$ 
7:     end for
8:     for all  $i, k$  do ▷ Maximization step
9:        $\Pi_{ik} \leftarrow \frac{\mathbf{E}[c_{ik}]}{\sum_{k'=1}^{n_i} \mathbf{E}[c_{ik'}]}$ 
10:    end for
11:     $LL \leftarrow \sum_{e \in E} \log P(e)$ 
12:  until  $LL - LL_0 < \epsilon$ 
13:  return  $LL, \Pi_{ik}$  for all  $i, k$ 
14: end function

```

The dynamic programming algorithm computes, for each example, the probability $P(g_i)$ of the subgoals $\{g_1, \dots, g_m\}$, also called the *inside probability*, and the value $Q(g_i)$ which is called the *outside probability*. These names derive from the fact that the algorithm generalizes the Inside–Outside algorithm for Probabilistic Context-Free Grammar [Baker, 1979]. It also generalizes the forward-backward algorithm used for parameter learning in HMMs by the Baum–Welch algorithm [Rabiner, 1989].

The inside probabilities are computed by procedure GET-INSIDE-PROBS shown in Algorithm 14 that is the same as function PRISM-PROB of Algorithm 3.

Outside probabilities instead are defined as

$$Q(g_i) = \frac{\partial P(q)}{\partial P(g_i)}$$

and are computed recursively from $i = 1$ to $i = m$ using an algorithm similar to Procedure CIRCP of Algorithm 5 for d-DNNF. The derivation of the recursive formulas is also similar. Suppose g_i appears in the ground program as

$$\begin{array}{rcl}
 b_1 \leftarrow g_i, W_{11} & \dots & b_1 \leftarrow g_i, W_{1i_1} \\
 & & \dots \\
 b_K \leftarrow g_i, W_{K1} & \dots & b_K \leftarrow g_i, W_{Ki_K}
 \end{array}$$

Algorithm 14 Procedure GET-INSIDE-PROBS: Computation of inside probabilities.

```

1: procedure GET-INSIDE-PROBS( $q$ )
2:   for all  $i, k$  do
3:      $P(msw(i, v_k)) \leftarrow \Pi_{ik}$ 
4:   end for
5:   for  $i \leftarrow m \rightarrow 1$  do
6:      $P(g_i) \leftarrow 0$ 
7:     for  $j \leftarrow 1 \rightarrow s_i$  do
8:       Let  $S_{ij}$  be  $h_{ij1}, \dots, h_{ij\sigma}$ 
9:        $P(S_{ij}) \leftarrow \prod_{l=1}^{\sigma} P(h_{ijl})$ 
10:       $P(g_i) \leftarrow P(g_i) + P(S_{ij})$ 
11:    end for
12:  end for
13: end procedure

```

and suppose the subgoals g_i may also be msw atoms. Then

$$P(b_1) = P(g_i, W_{11}) + \dots + P(g_i, W_{1i_1})$$

$$\dots$$

$$P(b_K) = P(g_i, W_{K1}) + \dots + P(g_i, W_{Ki_K})$$

We have that $Q(g_1) = 1$ as $q = g_1$. For $i = 2, \dots, m$, we can derive $Q(g_i)$ by the chain rule of the derivative knowing that $P(q)$ is a function of $P(b_1), \dots, P(b_K)$

$$Q(g_i) = \frac{\partial P(q)}{\partial P(b_1)} \frac{\partial P(g_i, W_{11})}{\partial P(g_1)} + \dots + \frac{\partial P(q)}{\partial P(b_K)} \frac{\partial P(g_i, W_{Ki_K})}{\partial P(g_1)} =$$

$$Q(b_1)P(g_i, W_{11})/P(g_i) + \dots + P(g_i, W_{Ki_K})/P(g_i)$$

This leads to the following recursive formula

$$Q(g_1) = 1$$

$$Q(g_i) = Q(b_1) \sum_{s=1}^{i_1} \frac{P(g_i, W_{1s})}{P(g_i)} + \dots + Q(b_K) \sum_{s=1}^{i_K} \frac{P(g_i, W_{Ks})}{P(g_i)}$$

that can be evaluated top-down from $q = g_1$ down to g_m . Procedure GET-OUTSIDE-PROBS of Algorithm 15 does this: for each subgoal b_k and each of its explanations g_i, W_{ks} , it updates the outside probability $Q(g_i)$ for the subgoal g_i .

If $g_i = msw(i, x_k)\theta_j$, then

$$P(X_{ij} = x_{ik}, e) = Q(g_i)P(g_i) = Q(g_i)\Pi_{ik}.$$

Algorithm 15 Procedure GET-OUTSIDE-PROBS: Computation of outside probabilities.

```

1: procedure GET-OUTSIDE-PROBS( $g$ )
2:    $Q(g_1) \leftarrow 1.0$ 
3:   for  $i \leftarrow 2 \rightarrow m$  do
4:      $Q(g_i) \leftarrow 0.0$ 
5:     for  $j \leftarrow 1 \rightarrow s_i$  do
6:       Let  $S_{ij}$  be  $h_{ij1}, \dots, h_{ij o}$ 
7:       for  $l \leftarrow 1 \rightarrow o$  do
8:          $Q(h_{ijl}) \leftarrow Q(h_{ijl}) + Q(g_i)P(S_{ij})/P(h_{ijl})$ 
9:       end for
10:    end for
11:  end for
12: end procedure

```

In fact, we can divide the explanations for e into two sets, K_{e1} , that includes the explanations containing $msw(i, x_k)\theta_j$, and K_{e2} , that includes the other explanations. Then $P(e) = P(K_{e1}) + P(K_{e2})$ and $P(X_{ij} = x_{ik}, e) = P(K_{e1})$. Since each explanation in K_{e1} contains $g_i = msw(i, x_k)\theta_j$, K_{e1} takes the form $\{\{g_i, W_1\}, \dots, \{g_i, W_s\}\}$ and

$$P(K_{e1}) = \sum_{\{g_i, W\} \in K_{e1}} P(g_i)P(W) = P(g_i) \sum_{\{g_i, W\} \in K_{e1}} P(W) \quad (9.1)$$

So we obtain

$$\begin{aligned}
 P(X_{ij} = x_{ik}, e) &= P(g_i) \sum_{\{g_i, W\} \in K_{e1}} P(W) = \\
 &= \frac{\partial P(K_e)}{\partial P(g_i)} P(g_i) = \\
 &= \frac{\partial P(e)}{\partial P(g_i)} P(g_i) = Q(g_i)P(g_i)
 \end{aligned} \quad (9.2)$$

where equality (9.2) holds because $\frac{\partial P(K_{e2})}{\partial P(g_i)} = 0$.

Function PRISM-EM of Algorithm 16 implements the overall EM algorithm [Sato and Kameya, 2001]. It calls procedure PRISM-EXPECTATION of Algorithm 17 that updates the expected values of the counters.

Algorithm 16 Function PRISM-EM.

```

1: function PRISM-EM( $E, \mathcal{P}, \epsilon$ )
2:    $LL = -inf$ 
3:   repeat
4:      $LL_0 = LL$ 
5:      $LL = \text{EXPECTATION}(E)$ 
6:     for all  $i$  do
7:        $Sum \leftarrow \sum_{k=1}^{n_i} \mathbf{E}[c_{ik}]$ 
8:       for  $k = 1$  to  $n_i$  do
9:          $\Pi_{ik} = \frac{\mathbf{E}[c_{ik}]}{Sum}$ 
10:      end for
11:    end for
12:  until  $LL - LL_0 < \epsilon$ 
13:  return  $LL, \Pi_{ik}$  for all  $i, k$ 
14: end function

```

Algorithm 17 Procedure PRISM-EXPECTATION.

```

1: function PRISM-EXPECTATION( $E$ )
2:    $LL = 0$ 
3:   for all  $e \in E$  do
4:     GET-INSIDE-PROBS( $e$ )
5:     GET-OUTSIDE-PROBS( $e$ )
6:     for all  $i$  do
7:       for  $k = 1$  to  $n_i$  do
8:          $\mathbf{E}[c_{ik}] = \mathbf{E}[c_{ik}] + Q(msw(i, x_k))\Pi_{ik}/P(e)$ 
9:       end for
10:    end for
11:     $LL = LL + \log P(e)$ 
12:  end for
13:  return  $LL$ 
14: end function

```

Sato and Kameya [2001] show that the combination of tabling with Algorithm 16 yields a procedure that has the same time complexity for programs encoding HMMs and PCFGs as the specific parameter learning algorithms: the Baum–Welch algorithm for HMMs [Rabiner, 1989] and the Inside–Outside algorithm for PCFGs [Baker, 1979].

9.2 LLPAD and ALLPAD Parameter Learning

The systems LLPAD [Riguzzi, 2004] and ALLPAD [Riguzzi, 2007b, 2008b] consider the problem of learning both the parameters and the structure of

LPADs from interpretations. We consider here parameter learning; we will discuss structure learning in Section 10.2.

Definition 47 (LLPAD Parameter learning problem). *Given a set*

$$E = \{(I, p_I) \mid I \in \text{Int}2, p_I \in [0, 1]\}$$

such that $\sum_{(I, p_I) \in E} p_I = 1$, find the value of the parameters of a ground LPAD \mathcal{P} , if they exist, such that

$$\forall (I, p_I) \in E : P(I) = p_I.$$

E may also be given as a multiset E' of interpretations. From this case, we can obtain a learning problem of the form above by computing a probability for each distinct interpretation in E' by relative frequency.

Notice that, if $\forall (I, p_I) \in E : P(I) = p_I$, then $\forall I \in \text{Int}2 : P(I) = p_I$ if we define $p_I = 0$ for those I not appearing in E , as $P(I)$ is a probability distribution over $\text{Int}2$ and $\sum_{(I, p_I) \in E} p_I = 1$.

Riguzzi [2004] presents a theorem that shows that, if all the couples of clauses of \mathcal{P} that share an atom in the head have mutually exclusive bodies, then the parameters can be computed by relative frequency.

Definition 48 (Mutually exclusive bodies). *Ground clauses $h_1 \leftarrow B_1$ and $h_2 \leftarrow B_2$ have mutually exclusive bodies over a set of interpretations J if, $\forall I \in J$, B_1 and B_2 are not both true in I .*

Mutual exclusivity of bodies is equivalent to the exclusive-or assumption.

Theorem 14 (Parameters as relative frequency). *Consider a ground locally stratified LPAD \mathcal{P} and a clause $C \in \mathcal{P}$ of the form*

$$C = h_1 : \Pi_1 ; h_2 : \Pi_2 ; \dots ; h_m : \Pi_m \leftarrow B.$$

Suppose all the clauses of \mathcal{P} that share an atom in the head with C have mutually exclusive bodies with C over the set of interpretations $\mathcal{J} = \{I \mid P(I) > 0\}$. In this case:

$$P(h_i \mid B) = \Pi_i$$

This theorem means that, under certain conditions, the probabilities in a clause's head can be interpreted as conditional probabilities of the head atoms given the body. Since $P(h_i|B) = P(h_i, B)/P(B)$, the probabilities of the head disjuncts of a ground rule can be computed from the probability distribution $P(I)$ defined by the program over the interpretations: for a set of literals S , $P(S) = \sum_{S \subseteq I} P(I)$. Moreover, since $\forall I \in \text{Int} : P(I) = p_I$, then $P(S) = \sum_{S \subseteq I} p_I$.

In fact, if the clauses have mutually exclusive bodies, there is no more uncertainty on the values of the hidden variables: for an atom in the head of a clause to be true, it must be selected by the only clause whose body is true. Therefore, there is no need of an EM algorithm and relative frequency can be used.

9.3 LeProbLog

LeProbLog [Gutmann et al., 2008] is a parameter learning system that starts from a set of examples annotated with a probability. The aim of LeProbLog is then to find the value of the parameters of a ProbLog program so that the probability assigned by the program to the examples is as close as possible to the one given.

Definition 49 (LeProbLog parameter learning problem). *Given a ProbLog program \mathcal{P} and a set of training examples $E = \{(e_1, p_1), \dots, (e_T, p_T)\}$ where e_t is a ground atom and $p_t \in [0, 1]$ for $t = 1, \dots, T$, find the parameter of the program so that the mean squared error*

$$MSE = \frac{1}{T} \sum_{t=1}^T (P(e_t) - p_t)^2$$

is minimized.

To perform learning, LeProbLog uses gradient descent, i.e., it iteratively updates the parameters in the opposite direction of the gradient. This requires the computation of the gradient which is

$$\frac{\partial MSE}{\partial \Pi_j} = \frac{2}{T} \sum_{t=1}^T (P(e_t) - p_t) \cdot \frac{\partial P(e_t)}{\partial \Pi_j}$$

LeProbLog compiles queries to BDDs; therefore, $P(e_t)$ can be computed with Algorithm 4. To compute $\frac{\partial P(e_t)}{\partial \Pi_j}$, it uses a dynamic programming

algorithm that traverses the BDD bottom up. In fact

$$\frac{\partial P(e_t)}{\partial \Pi_j} = \frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j}$$

where $f(\mathbf{X})$ is the Boolean function represented by the BDD. $f(\mathbf{X})$ is

$$f(\mathbf{X}) = X_k \cdot f^{X_k}(\mathbf{X}) + \neg X_k \cdot f^{\neg X_k}(\mathbf{X})$$

where X_k is the random Boolean variable associated with the root and to ground fact $\Pi_k :: f_k$, so

$$P(f(\mathbf{X})) = \Pi_k \cdot P(f^{X_k}(\mathbf{X})) + (1 - \Pi_k) \cdot P(f^{\neg X_k}(\mathbf{X}))$$

and

$$\frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j} = P(f^{X_k}(\mathbf{X})) - P(f^{\neg X_k}(\mathbf{X}))$$

if $k = j$, or

$$\frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j} = \Pi_k \cdot \frac{\partial P(f^{X_k}(\mathbf{X}))}{\partial \Pi_j} + (1 - \Pi_k) \cdot \frac{\partial P(f^{\neg X_k}(\mathbf{X}))}{\partial \Pi_j}$$

if $k \neq j$. Moreover

$$\frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j} = 0$$

if X_j does not appear in \mathbf{X} .

When performing gradient descent, we have to ensure that the parameters remain in the $[0, 1]$ interval. However, updating the parameters using the gradient does not guarantee that. Therefore, a reparameterization is used by means of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ that takes a real value $x \in (-\infty, +\infty)$ and returns a real value in $(0, 1)$. So each parameter is expressed as $\Pi_j = \sigma(a_j)$ and the a_j s are used as the parameters to update. Since $a_j \in (-\infty, +\infty)$, we do not risk to get values outside the domain.

Given that $\frac{d\sigma(x)}{dx} = \sigma(x) \cdot (1 - \sigma(x))$, using the chain rule of derivatives, we get

$$\frac{\partial P(e_t)}{\partial a_j} = \sigma(a_j) \cdot (1 - \sigma(a_j)) \frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j}$$

LeProbLog dynamic programming function for computing $\frac{\partial P(f(\mathbf{X}))}{\partial a_j}$ is shown in Algorithm 18. `GRADIENTEVAL(n, j)` traverses the BDD n and returns two values: a real number and a Boolean variable *seen* which is 1 if the variable X_j was seen in n . We consider three cases:

Algorithm 18 Function GRADIENT.

```

1: function GRADIENT( $BDD, j$ )
2:   ( $val, seen$ )  $\leftarrow$  GRADIENTEVAL( $BDD, j$ )
3:   if  $seen = 1$  then
4:     return  $val \cdot \sigma(a_j) \cdot (1 - \sigma(a_j))$ 
5:   else
6:     return 0
7:   end if
8: end function
9: function GRADIENTEVAL( $n, j$ )
10:  if  $n$  is the 1-terminal then
11:    return (1, 0)
12:  end if
13:  if  $n$  is the 0-terminal then
14:    return (0, 0)
15:  end if
16:  ( $val(child_1(n)), seen(child_1(n))$ )  $\leftarrow$  GRADIENTEVAL( $child_1(n), j$ )
17:  ( $val(child_0(n)), seen(child_0(n))$ )  $\leftarrow$  GRADIENTEVAL( $child_0(n), j$ )
18:  if  $varindex(n) = j$  then
19:    return ( $val(child_1(n)) - val(child_0(n)), 1$ )
20:  else if  $seen(child_1(n)) = seen(child_0(n))$  then
21:    return ( $\sigma(a_n) \cdot val(child_1(n)) + (1 - \sigma(a_n)) \cdot val(child_0(n)), seen(child_1(n))$ )
22:  else if  $seen(child_1(n)) = 1$  then
23:    return ( $\sigma(a_n) \cdot val(child_1(n)), 1$ )
24:  else if  $seen(child_0(n)) = 1$  then
25:    return ( $(1 - \sigma(a_n)) \cdot val(child_0(n)), 1$ )
26:  end if
27: end function

```

1. If the variable of node n is below X_j in the BDD order, then GRADIENTEVAL returns the probability of node n and $seen = 0$.
2. If the variable of node n is X_j , then GRADIENTEVAL returns $seen = 1$ and the gradient given by the difference of the values of its two children $val(child_1(n)) - val(child_0(n))$.
3. If the variable of node n is above X_j in the BDD order, then GRADIENTEVAL returns $\sigma(a_n) \cdot val(child_1(n)) + (1 - \sigma(a_n)) \cdot val(child_0(n))$ unless X_j does not appear in one of the sub-BDD, in which case the corresponding term is 0.

GRADIENTEVAL determines which of the cases applies by using function $varindex(n)$ that returns the index of the variable of node n and by considering the values $seen(child_1(n))$ and $seen(child_0(n))$ of the children: if one of them is 1 and the other is 0, then X_j is below the variable of node n and we

fall in the third case above. If they are both 1, we are in the third case again. If they are both 0, we are either in the first case or in third case but X_j does not appear in the BDD. We deal with the latter situation by returning 0 in the outer function GRADIENT.

The overall LeProbLog function is shown in Algorithm 19. Given a ProbLog program \mathcal{P} with n probabilistic ground facts, it returns the values of their parameters. It initializes the vector of parameters $\mathbf{a} = (a_1, \dots, a_n)$ randomly and then computes an update $\Delta\mathbf{a}$ by computing the gradient. \mathbf{a} is then updated by subtracting $\Delta\mathbf{a}$ multiplied by a learning rate η .

Algorithm 19 Function LEPROBLOG: LeProbLog algorithm.

```

1: function LEPROBLOG( $E, \mathcal{P}, k, \eta$ )
2:   initialize all  $a_j$  randomly
3:   while not converged do
4:      $\Delta\mathbf{a} \leftarrow 0$ 
5:     for  $t \leftarrow 1 \rightarrow T$  do
6:       find  $k$  best proofs and generate  $BDD_t$  for  $e_t$ 
7:        $y \leftarrow \frac{2}{T}(P(e_t) - p_t)$ 
8:       for  $j \leftarrow 1 \rightarrow n$  do
9:          $deriv_j \leftarrow \text{GRADIENT}(BDD_t, j)$ 
10:         $\Delta a_j \leftarrow \Delta a_j + y \cdot deriv_j$ 
11:       end for
12:     end for
13:      $\mathbf{a} \leftarrow \mathbf{a} - \eta \cdot \Delta\mathbf{a}$ 
14:   end while
15:   return  $\{\sigma(a_1), \dots, \sigma(a_n)\}$ 
16: end function

```

The BDDs for examples are built by computing the k best explanations for each example, as in the k -best inference algorithm, see Section 7.1.2. As the set of the k best explanations may change when the parameters change, the BDDs are recomputed at each iteration.

9.4 EMBLEM

EMBLEM [Bellodi and Riguzzi, 2013, 2012] applies the algorithm for performing EM over BDDs proposed in [Thon et al., 2008; Ishihata et al., 2008a,b; Inoue et al., 2009] to the problem of learning the parameters of an LPAD.

Definition 50 (EMBLEM parameter learning problem). *Given an LPAD \mathcal{P} with unknown parameters and two sets $E^+ = \{e_1, \dots, e_T\}$ and $E^- =$*

$\{e_{T+1}, \dots, e_Q\}$ of ground atoms (positive and negative examples), find the value of the parameters Π of \mathcal{P} that maximize the likelihood of the examples, i.e., solve

$$\arg \max_{\Pi} P(E^+, \sim E^-) = \arg \max_{\Pi} \prod_{t=1}^T P(e_t) \prod_{t=T+1}^Q P(\sim e_t).$$

The predicates for the atoms in E^+ and E^- are called target because the objective is to be able to better predict the truth value of atoms for them.

Typically, the LPAD \mathcal{P} has two components: a set of rules, annotated with parameters and representing general knowledge, and a set of certain ground facts, representing background knowledge on individual cases of a specific world, from which consequences can be drawn with the rules, including the examples. Sometimes, it is useful to provide information on more than one world. For each world, a background knowledge and sets of positive and negative examples are provided. The description of one world is also called a *mega-interpretation* or *mega-example*. In this case, it is useful to encode the positive examples as ground facts of the mega-interpretation and the negative examples as suitably annotated ground facts (such as $neg(a)$ for negative example a) for one or more target predicates. The task then is maximizing the product of the likelihood of the examples for all mega-interpretations.

EMBLEM generates a BDD for each example in $E = \{e_1, \dots, e_T, \sim e_{T+1}, \dots, \sim e_Q\}$. The BDD for an example e encodes its explanations. Then EMBLEM enters the EM cycle, in which the steps of expectation and maximization are repeated until the log likelihood of the examples reaches a local maximum.

Let us now present the formulas for the expectation and maximization phases. EMBLEM adopts the encoding of multivalued random variable with Boolean random variables used in PITA, see Section 5.6. Let X_{ijk} for $k = 1, \dots, n_i - 1$ and $j \in g(i)$ be the Boolean random variables associated with grounding $C_i\theta_j$ of clause C_i of \mathcal{P} where n_i is the number of head atoms of C_i and $g(i)$ is the set of indices of grounding substitutions of C_i .

Example 99 (Epidemic – LPAD – EM). *Let us recall Example 87 about the development of an epidemic*

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

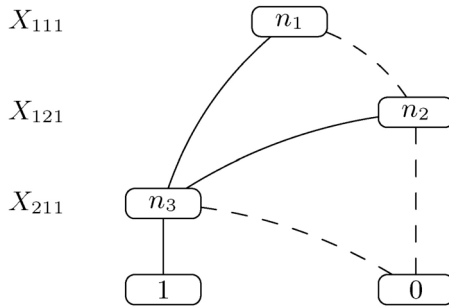


Figure 9.1 BDD for query *epidemic* for Example 99. From [Bellodi and Riguzzi, 2013].

Clause C_1 has two groundings, both with three atoms in the head, the first associated with Boolean random variables X_{111} and X_{112} and the latter with X_{121} and X_{122} . C_2 has a single grounding with two atoms in the head and is associated with variable X_{211} . The BDD for query *epidemic* is shown in Figure 9.1.

The EM algorithm alternates between the two phases:

- **Expectation:** computes $\mathbf{E}[c_{ik0}|e]$ and $\mathbf{E}[c_{ik1}|e]$ for all examples e , rules C_i in \mathcal{P} and $k = 1, \dots, n_i - 1$, where c_{ikx} is the number of times a variable X_{ijk} takes value x for $x \in \{0, 1\}$, with j in $g(i)$. $\mathbf{E}[c_{ikx}|e]$ is given by $\sum_{j \in g(i)} P(X_{ijk} = x|e)$.
- **Maximization:** computes π_{ik} for all rules C_i and $k = 1, \dots, n_i - 1$ as

$$\pi_{ik} = \frac{\sum_{e \in E} \mathbf{E}[c_{ik1}|e]}{\sum_{q \in E} \mathbf{E}[c_{ik0}|e] + \mathbf{E}[c_{ik1}|e]}.$$

$P(X_{ijk} = x|e)$ is given by $P(X_{ijk} = x|e) = \frac{P(X_{ijk}=x,e)}{P(e)}$.

Now consider a BDD for an example e built by applying only the merge rule, fusing together identical sub-diagrams but not deleting nodes. For example, by applying only the merge rule in Example 99, the diagram in Figure 9.2 is obtained. The resulting diagram, that we call Complete Binary Decision Diagram (CBDD), is such that every path contains a node for every level.

$P(e)$ is given by the sum of the probabilities of all the paths in the CBDD from the root to a 1 leaf, where the probability of a path is defined as the product of the probabilities of the individual choices along the path. Variable X_{ijk} is associated with a level l in the sense that all nodes at that level test variable X_{ijk} . All paths from the root to a leaf pass through a node of level l .

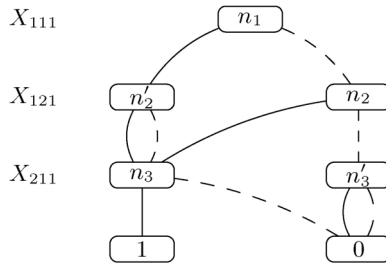


Figure 9.2 BDD after applying the merge rule only for Example 99. From [Bellodi and Riguzzi, 2013].

We can express $P(e)$ as

$$P(e) = \sum_{\rho \in R(e)} \prod_{d \in \rho} \pi(d)$$

where $R(e)$ is the set of paths for query e that lead to a 1 leaf, d is an edge of path ρ , and $\pi(d)$ is the probability associated with the edge: if d is the 1-branch from a node associated with a variable X_{ijk} , then $\pi(d) = \pi_{ik}$; if d is the 0-branch from a node associated with a variable X_{ijk} , then $\pi(d) = 1 - \pi_{ik}$.

We can further expand $P(e)$ as

$$P(e) = \sum_{n \in N(X_{ijk}), \rho \in R(e)} \pi_{ikx} \prod_{d \in \rho^{n,x}} \pi(d) \prod_{d \in \rho_n} \pi(d)$$

where $N(X_{ijk})$ is the set of nodes associated with variable X_{ijk} , ρ_n is the portion of path ρ up to node n , $\rho^{n,x}$ is the portion of path ρ from $child_x(n)$ to the 1 leaf, and π_{ikx} is π_{ik} if $x = 1$ and $1 - \pi_{ik}$ otherwise. Then

$$P(e) = \sum_{n \in N(X_{ijk}), \rho_n \in R_n(q), x \in \{0,1\}} \pi_{ikx} \prod_{d \in \rho^{n,x}} \pi(d) \prod_{d \in \rho_n} \pi(d)$$

where $R_n(q)$ is the set containing the paths from the root to n and $R^n(q, x)$ is the set of paths from $child_x(n)$ to the 1 leaf.

To compute $P(X_{ijk} = x, e)$, we can observe that we need to consider only the paths passing through the x -child of a node n associated with variable X_{ijk} , so

$$P(X_{ijk} = x, e) = \sum_{n \in N(X_{ijk}), \rho_n \in R_n(q), \rho^n \in R^n(q, x)} \pi_{ikx} \prod_{d \in \rho^n} \pi(d) \prod_{d \in \rho_n} \pi(d)$$

We can rearrange the terms in the summation as

$$\begin{aligned}
 P(X_{ijk} = x, e) &= \sum_{n \in N(X_{ijk})} \sum_{\rho_n \in R_n(q)} \sum_{\rho^n \in R^n(q, x)} \pi_{ikx} \prod_{d \in \rho^n} \pi(d) \prod_{d \in \rho_n} \pi(d) \\
 &= \sum_{n \in N(X_{ijk})} \pi_{ikx} \sum_{\rho_n \in R_n(q)} \prod_{d \in \rho_n} \pi(d) \sum_{\rho^n \in R^n(q, x)} \prod_{d \in \rho^n} \pi(d) \\
 &= \sum_{n \in N(X_{ijk})} \pi_{ikx} F(n) B(\text{child}_x(n))
 \end{aligned}$$

where $F(n)$ is the *forward probability* [Ishihata et al., 2008b], the probability mass of the paths from the root to n , while $B(n)$ is the *backward probability* [Ishihata et al., 2008b], the probability mass of paths from n to the 1 leaf. If $root$ is the root of a tree for a query e , then $B(root) = P(e)$.

The expression $\pi_{ikx} F(n) B(\text{child}_x(n))$ represents the sum of the probability of all the paths passing through the x -edge of node n . We indicate with $e^x(n)$ such an expression. Thus

$$P(X_{ijk} = x, e) = \sum_{n \in N(X_{ijk})} e^x(n) \quad (9.3)$$

For the case of a BDD, i.e., a diagram obtained by also applying the deletion rule, Equation (9.3) is no longer valid since paths where there is no node associated with X_{ijk} can also contribute to $P(X_{ijk} = x, e)$. In fact, it is necessary to also consider the deleted paths: suppose a node n associated with variable Y has a level higher than variable X_{ijk} and suppose that $\text{child}_0(n)$ is associated with variable W that has a level lower than variable X_{ijk} . The nodes associated with variable X_{ijk} have been deleted from the paths from n to $\text{child}_0(n)$. One can imagine that the current BDD has been obtained from a BDD having a node m associated with variable X_{ijk} that is a descendant of n along the 0-branch and whose outgoing edges both point to $\text{child}_0(n)$. The probability mass of the two paths that were merged was $e^0(n)(1 - \pi_{ik})$ and $e^0(n)\pi_{ik}$ for the paths passing through the 0-child and 1-child of m respectively. The first quantity contributes to $P(X_{ijk} = 0, e)$ and the latter to $P(X_{ijk} = 1, e)$.

Formally, let $Del^x(X)$ be the set of nodes n such that the level of X is below that of n and is above that of $\text{child}_x(n)$, i.e., X is deleted between n and $\text{child}_x(n)$. For the BDD in Figure 9.1, for example, $Del^1(X_{121}) = \{n_1\}$,

$Del^0(X_{121}) = \emptyset$, $Del^1(X_{221}) = \emptyset$, and $Del^0(X_{221}) = \{n_3\}$. Then

$$P(X_{ijk} = 0|e) = \sum_{n \in N(X_{ijk})} e^x(n) + (1 - \pi_{ik}) \left(\sum_{n \in Del^0(X_{ijk})} e^0(n) + \sum_{n \in Del^1(X_{ijk})} e^1(n) \right)$$

$$P(X_{ijk} = 1, e) = \sum_{n \in N(X_{ijk})} e^x(n) + \pi_{ik} \left(\sum_{n \in Del^0(X_{ijk})} e^0(n) + \sum_{n \in Del^1(X_{ijk})} e^1(n) \right)$$

Having shown how to compute the probabilities, we now describe EMBLEM in detail. The typical input for EMBLEM will be a set of mega-interpretations, i.e., sets of ground facts, each describing a portion of the domain of interest. Among the predicates for the input facts, the user has to indicate which are target predicates: the facts for these predicates will then form the examples, i.e., the queries for which the BDDs are built. The predicates can be treated as closed-world or open-world. In the first case, a closed-world assumption is made, so the body of clauses with a target predicate in the head is resolved only with facts in the interpretation. In the second case, the body of clauses with a target predicate in the head is resolved both with facts in the interpretation and with clauses in the theory. If the last option is set and the theory is cyclic, EMBLEM uses a depth bound on SLD derivations to avoid going into infinite loops, as proposed by [Gutmann et al., 2010].

EMBLEM, shown in Algorithm 20, consists of a cycle in which the procedures EXPECTATION and MAXIMIZATION are repeatedly called. Procedure EXPECTATION returns the LL of the data that is used in the stopping criterion: EMBLEM stops when the difference between the LL of the current and previous iteration drops below a threshold ϵ or when this difference is below a fraction δ of the current LL.

Procedure EXPECTATION, shown in Algorithm 21, takes as input a list of BDDs, one for each example, and computes the expectation for each one, i.e., $P(e, X_{ijk} = x)$ for all variables X_{ijk} in the BDD. In the procedure, we use $\eta^x(i, k)$ to indicate $\sum_{j \in g(i)} P(e, X_{ijk} = x)$. EXPECTATION first

calls GETFORWARD and GETBACKWARD that compute the forward, the backward probability of nodes and $\eta^x(i, k)$ for non-deleted paths only. Then it updates $\eta^x(i, k)$ to take into account deleted paths.

Algorithm 20 Function EMBLEM.

```

1: function EMBLEM( $E, \mathcal{P}, \epsilon, \delta$ )
2:   build  $BDDs$ 
3:    $LL = -inf$ 
4:   repeat
5:      $LL_0 = LL$ 
6:      $LL = \text{EXPECTATION}(BDDs)$ 
7:     MAXIMIZATION
8:   until  $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta$ 
9:   return  $LL, \pi_{ik}$  for all  $i, k$ 
10: end function

```

Algorithm 21 Function EXPECTATION.

```

1: function EXPECTATION( $BDDs$ )
2:    $LL = 0$ 
3:   for all  $BDD \in BDDs$  do
4:     for all  $i$  do
5:       for  $k = 1$  to  $n_i - 1$  do
6:          $\eta^0(i, k) = 0; \eta^1(i, k) = 0$ 
7:       end for
8:     end for
9:     for all variables  $X$  do
10:       $\zeta(X) = 0$ 
11:    end for
12:    GETFORWARD( $root(BDD)$ )
13:     $Prob = \text{GETBACKWARD}(root(BDD))$ 
14:     $T = 0$ 
15:    for  $l = 1$  to  $levels(BDD)$  do
16:      Let  $X_{ijk}$  be the variable associated with level  $l$ 
17:       $T = T + \zeta(X_{ijk})$ 
18:       $\eta^0(i, k) = \eta^0(i, k) + T \times (1 - \pi_{ik})$ 
19:       $\eta^1(i, k) = \eta^1(i, k) + T \times \pi_{ik}$ 
20:    end for
21:    for all  $i$  do
22:      for  $k = 1$  to  $n_i - 1$  do
23:         $\mathbf{E}[c_{ik0}] = \mathbf{E}[c_{ik0}] + \eta^0(i, k)/Prob$ 
24:         $\mathbf{E}[c_{ik1}] = \mathbf{E}[c_{ik1}] + \eta^1(i, k)/Prob$ 
25:      end for
26:    end for
27:     $LL = LL + \log(Prob)$ 
28:  end for
29:  return  $LL$ 
30: end function

```

Algorithm 22 Procedure MAXIMIZATION.

```

1: procedure MAXIMIZATION
2:   for all  $i$  do
3:     for  $k = 1$  to  $n_i - 1$  do
4:        $\pi(ik) = \frac{E[c_{ik1}]}{E[c_{ik0}] + E[c_{ik1}]}$ 
5:     end for
6:   end for
7: end procedure

```

Procedure MAXIMIZATION (Algorithm 22) computes the parameters values for the next EM iteration.

Procedure GETFORWARD, shown in Algorithm 23, computes the value of the forward probabilities. It traverses the diagram one level at a time starting from the root level. For each level, it considers each node n and computes its contribution to the forward probabilities of its children. Then the forward probabilities of its children, stored in table F , are updated.

Algorithm 23 Procedure GETFORWARD: Computation of the forward probability.

```

1: procedure GETFORWARD( $root$ )
2:    $F(root) = 1$ 
3:    $F(n) = 0$  for all nodes
4:   for  $l = 1$  to  $levels$  do            $\triangleright levels$  is the number of levels of the BDD rooted at  $root$ 
5:      $Nodes(l) = \emptyset$ 
6:   end for
7:    $Nodes(1) = \{root\}$ 
8:   for  $l = 1$  to  $levels$  do
9:     for all  $node \in Nodes(l)$  do
10:      let  $X_{ijk}$  be  $v(node)$ , the variable associated with  $node$ 
11:      if  $child_0(node)$  is not terminal then
12:         $F(child_0(node)) = F(child_0(node)) + F(node) \cdot (1 - \pi_{ik})$ 
13:        add  $child_0(node)$  to  $Nodes(level(child_0(node)))$     $\triangleright level(node)$  returns the
        level of  $node$ 
14:      end if
15:      if  $child_1(node)$  is not terminal then
16:         $F(child_1(node)) = F(child_1(node)) + F(node) \cdot \pi_{ik}$ 
17:        add  $child_1(node)$  to  $Nodes(level(child_1(node)))$ 
18:      end if
19:    end for
20:  end for
21: end procedure

```

Function GETBACKWARD, shown in Algorithm 24, computes the backward probability of nodes by traversing recursively the tree from the root to the leaves. When the calls of GETBACKWARD for both children of a node n

return, we have all the information that is needed to compute the e^x values and the value of $\eta^x(i, k)$ for non-deleted paths.

Algorithm 24 Procedure GETBACKWARD: Computation of the backward probability, updating of η and of ς .

```

1: function GETBACKWARD(node)
2:   if node is a terminal then
3:     return value(node)
4:   else
5:     let  $X_{ijk}$  be  $v(\textit{node})$ 
6:      $B(\textit{child}_0(\textit{node})) = \text{GETBACKWARD}(\textit{child}_0(\textit{node}))$ 
7:      $B(\textit{child}_1(\textit{node})) = \text{GETBACKWARD}(\textit{child}_1(\textit{node}))$ 
8:      $e^0(\textit{node}) = F(\textit{node}) \cdot B(\textit{child}_0(\textit{node})) \cdot (1 - \pi_{ik})$ 
9:      $e^1(\textit{node}) = F(\textit{node}) \cdot B(\textit{child}_1(\textit{node})) \cdot \pi_{ik}$ 
10:     $\eta^0(i, k) = \eta_t^0(i, k) + e^0(\textit{node})$ 
11:     $\eta^1(i, k) = \eta_t^1(i, k) + e^1(\textit{node})$ 
12:     $V\textit{Succ} = \textit{succ}(v(\textit{node}))$   $\triangleright \textit{succ}(X)$  returns the variable following  $X$  in the order
13:     $\varsigma(V\textit{Succ}) = \varsigma(V\textit{Succ}) + e^0(\textit{node}) + e^1(\textit{node})$ 
14:     $\varsigma(v(\textit{child}_0(\textit{node}))) = \varsigma(v(\textit{child}_0(\textit{node}))) - e^0(\textit{node})$ 
15:     $\varsigma(v(\textit{child}_1(\textit{node}))) = \varsigma(v(\textit{child}_1(\textit{node}))) - e^1(\textit{node})$ 
16:    return  $B(\textit{child}_0(\textit{node})) \cdot (1 - \pi_{ik}) + B(\textit{child}_1(\textit{node})) \cdot \pi_{ik}$ 
17:   end if
18: end function

```

The array ς stores, for every variable X_{ijk} , an algebraic sum of $e^x(n)$: those for nodes in upper levels that do not have a descendant in the level l of X_{ijk} minus those for nodes in upper levels that have a descendant in level l . In this way, it is possible to add the contributions of the deleted paths by starting from the root level and accumulating $\varsigma(X_{ijk})$ for the various levels in a variable T (see lines 15–20 of Algorithm 21): an $e^x(n)$ value that is added to the accumulator T for the level of X_{ijk} means that n is an ancestor for nodes in that level. When the x -branch from n reaches a node in a level $l' \leq l$, $e^x(n)$ is subtracted from the accumulator, as it is not relative to a deleted node on the path anymore, see lines 14 and 15 of Algorithm 24.

Let us see an example of execution. Consider the program of Example 99 and the single example *epidemic*. The BDD of Figure 9.1 (also shown in Figure 9.3) is built and passed to EXPECTATION in the form of a pointer to its root node n_1 . After initializing the η counters to 0, GETFORWARD is called with argument n_1 . The F table for n_1 is set to 1 since this is the root. Then F is computed for the 0-child, n_2 , as $0 + 1 \cdot 0.4 = 0.4$ and n_2 is added to $Nodes(2)$, the set of nodes for the second level. Then F is computed for the 1-child, n_3 , as $0 + 1 \cdot 0.6 = 0.6$, and n_3 is added to $Nodes(3)$. At the next iteration of the cycle, level 2 is considered and node n_2 is fetched from $Nodes(2)$. The 0-child is a terminal, so it is skipped, while the 1-child is n_3

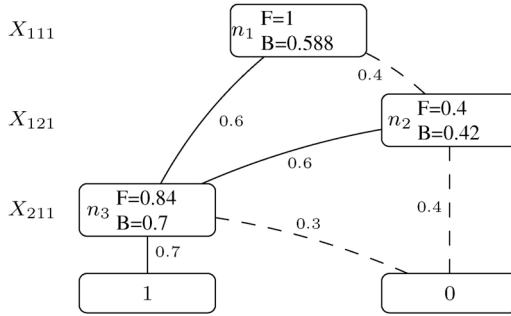


Figure 9.3 Forward and backward probabilities. F indicates the forward probability and B the backward probability of each node. From [Bellodi and Riguzzi, 2013].

and its F value is updated as $0.6 + 0.4 \cdot 0.6 = 0.84$. In the third iteration, node n_3 is fetched but, since its children are leaves, F is not updated. The resulting forward probabilities are shown in Figure 9.3.

Then `GETBACKWARD` is called on n_1 . The function calls `GETBACKWARD(n_2)` that in turn calls `GETBACKWARD(0)`. This call returns 0 because it is a terminal node. Then `GETBACKWARD(n_2)` calls `GETBACKWARD(n_3)` that in turn calls `GETBACKWARD(1)` and `GETBACKWARD(0)`, returning respectively 1 and 0. Then `GETBACKWARD(n_3)` computes $e^0(n_3)$ and $e^1(n_3)$ in the following way:

$$e^0(n_3) = F(n_3) \cdot B(0) \cdot 0.3(1 - \pi_{21}) = 0.84 \cdot 0 \cdot 0.3 = 0$$

$$e^1(n_3) = F(n_3) \cdot B(1) \cdot 0.7(\pi_{21}) = 0.84 \cdot 1 \cdot 0.7 = 0.588.$$

Now the counters for clause C_2 are updated:

$$\eta^0(2, 1) = 0$$

$$\eta^1(2, 1) = 0.588$$

while we do not show the update of ς since its value for the level of the leaves is not used afterward. `GETBACKWARD(n_3)` now returns the backward probability of n_3 $B(n_3) = 1 \cdot 0.7 + 0 \cdot 0.3 = 0.7$. `GETBACKWARD(n_2)` can proceed to compute

$$e^0(n_2) = F(n_2) \cdot B(0) \cdot 0.4(1 - \pi_{11}) = 0.4 \cdot 0.0 \cdot 0.4 = 0$$

$$e^1(n_2) = F(n_2) \cdot B(n_3) \cdot 0.6(\pi_{11}) = 0.4 \cdot 0.7 \cdot 0.6 = 0.168,$$

and $\eta^0(1, 1) = 0$, $\eta^1(1, 1) = 0.168$. The variable following X_{121} is X_{211} , so $\varsigma(X_{211}) = e^0(n_2) + e^1(n_2) = 0 + 0.168 = 0.168$. Since X_{121} is also associated with the 1-child n_2 , then $\varsigma(X_{211}) = \varsigma(X_{211}) - e^1(n_2) = 0$. The 0-child is a leaf so ς is not updated.

`GETBACKWARD(n_2)` then returns $B(n_2) = 0.7 \cdot 0.6 + 0 \cdot 0.4 = 0.42$ to `GETBACKWARD(n_1)` that computes $e^0(n_1)$ and $e^1(n_1)$ as

$$e^0(n_1) = F(n_1) \cdot B(n_2) \cdot 0.4(1 - \pi_{11}) = 1 \cdot 0.42 \cdot 0.4 = 0.168$$

$$e^1(n_1) = F(n_1) \cdot B(n_3) \cdot 0.6(\pi_{11}) = 1 \cdot 0.7 \cdot 0.6 = 0.42$$

and updates the η counters as $\eta^0(1, 1) = 0.168$, $\eta^1(1, 1) = 0.168 + 0.42 = 0.588$.

Finally, ς is updated:

$$\varsigma(X_{121}) = e^0(n_1) + e^1(n_1) = 0.168 + 0.42 = 0.588$$

$$\varsigma(X_{121}) = \varsigma(X_{121}) - e^0(n_1) = 0.42$$

$$\varsigma(X_{211}) = \varsigma(X_{211}) - e^1(n_1) = -0.42$$

GETBACKWARD(n_1) returns $B(n_1) = 0.7 \cdot 0.6 + 0.42 \cdot 0.4 = 0.588$ to EXPECTATION, that adds the contribution of deleted nodes by cycling over the BDD levels and updating T . Initially, T is set to 0, and then, for variable X_{111} , T is updated to $T = \varsigma(X_{111}) = 0$ which implies no modification of $\eta^0(1, 1)$ and $\eta^1(1, 1)$. For variable X_{121} , T is updated to $T = 0 + \varsigma(X_{121}) = 0.42$ and the η table is modified as

$$\eta^0(1, 1) = 0.168 + 0.42 \cdot 0.4 = 0.336$$

$$\eta^1(1, 1) = 0.588 + 0.42 \cdot 0.6 = 0.84$$

For variable X_{211} , T becomes $0.42 + \varsigma(X_{211}) = 0$, so $\eta^0(2, 1)$ and $\eta^1(2, 1)$ are not updated. At this point, the expected counts for the two rules can be computed

$$\mathbf{E}[c_{110}] = 0 + 0.336/0.588 = 0.5714285714$$

$$\mathbf{E}[c_{111}] = 0 + 0.84/0.588 = 1.4285714286$$

$$\mathbf{E}[c_{210}] = 0 + 0/0.588 = 0$$

$$\mathbf{E}[c_{211}] = 0 + 0.588/0.588 = 1$$

9.5 ProbLog2 Parameter Learning

ProbLog2 [Fierens et al., 2015] includes the algorithm LFI-ProbLog [Gutmann et al., 2011b] that learns the parameters of ProbLog programs from partial interpretations.

Partial interpretations are three valued interpretations: they specify the truth value of some but not necessarily all ground atoms. A partial interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ states that the atoms in I_T are true and those in I_F are false. A partial interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ can be associated with a conjunction $q(\mathcal{I}) = \bigwedge_{a \in I_T} a \wedge \bigwedge_{a \in I_F} \sim a$.

Definition 51 (LFI-ProbLog learning problem). *Given a ProbLog program \mathcal{P} with unknown parameters and a set $E = \{\mathcal{I}_1, \dots, \mathcal{I}_T\}$ of partial interpretations (the examples), find the value of the parameters $\mathbf{\Pi}$ of \mathcal{P} that maximize the likelihood of the examples, i.e., solve*

$$\arg \max_{\Pi} P(E) = \arg \max_{\Pi} \prod_{t=1}^T P(q(\mathcal{I}_t))$$

If all interpretations in E are total, Theorem 14 can be applied and the parameters can be computed by relative frequency, see Section 9.2. If some interpretations in E are partial, instead, an EM algorithm must be used, similar to the one used by PRISM and EMBLEM.

LFI-ProbLog generates a d-DNNF circuit for each partial interpretation $\mathcal{I} = \langle I_T, I_F \rangle$ by using the ProbLog2 algorithm of Section 5.7 with the evidence $q(\mathcal{I})$.

Then it associates a Boolean random variable X_{ij} with each ground probabilistic fact $f_i \theta_j$. For each example \mathcal{I} , variable X_{ij} , and $x \in \{0, 1\}$, LFI-ProbLog computes $P(X_{ij} = x | \mathcal{I})$. Then it uses this to compute $\mathbf{E}[c_{ix} | \mathcal{I}]$, the expected value given example \mathcal{I} of the number of times variable X_{ij} takes value x for any j in $g(i)$, the set of grounding substitutions of f_i . $\mathbf{E}[c_{ix}]$ is the expected value given all the examples. As in PRISM and EMBLEM, these are given by:

$$\mathbf{E}[c_{ix}] = \sum_{t=1}^T \mathbf{E}[c_{ix} | \mathcal{I}_t]$$

and

$$\mathbf{E}[c_{ix} | \mathcal{I}_t] = \sum_{j \in g(i)} P(X_{ij} = x | \mathcal{I}_t).$$

In the maximization phase, the parameter π_i of probabilistic fact f_i can be computed as

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

LFI-ProbLog computes $P(X_{ij} = x | \mathcal{I})$ by computing $P(X_{ij} = x, \mathcal{I})$ using Procedure CIRCP shown in Algorithm 5: the d-DNNF circuit is visited twice, once bottom up to compute $P(q(\mathcal{I}))$ and once top down to compute $P(X_{ij} = x | \mathcal{I})$ for all the variables X_{ij} and values x . Then $P(X_{ij} = x | \mathcal{I})$ is given by $\frac{P(X_{ij}=x, \mathcal{I})}{P(\mathcal{I})}$.

Nishino et al. [2014] extended LFI-ProbLog in order to perform sparse parameter learning, i.e., parameter learning while trying to reduce the number of parameters different from 0 or 1, in order to obtain a simpler program. To do so, they add a penalty term to the objective function and use gradient descent to optimize the parameters.

9.6 Parameter Learning for Hybrid Programs

Gutmann [2011] proposes an approach for learning the parameters of hybrid ProbLog programs that is based on the LeProbLog algorithm described in Section 9.3. In particular, Gutmann [2011] shows how the gradient of the objective function can also be computed for hybrid ProbLog programs.

Islam [2012]; Islam et al. [2012a] present a parameter learning algorithm for Extended PRISM. The algorithm is based on PRISM's EM procedure and involves the computation of the Expected Sufficient Statistics (ESS) of the random variables. The ESS for discrete random variables are represented by a tuple of the expected counts of each values of the variable.

The ESS of a Gaussian random variable X are the triple

$$(ESS^X, ESS^{X^2}, ESS^{count})$$

where the components denote the expected sum, expected sum of squares, and the expected number of uses of random variable X , respectively.

ESS can be computed by deriving examples individually. A quicker approach consists in building a symbolic derivation, computing ESS functions instead of plain ESS, and then applying them to each example. The derivation of ESS functions is similar to the one of success functions for Extended PRISM discussed in Section 5.11.

10

Structure Learning

The techniques presented in this chapter aim at inducing whole programs from data, including both their structure and parameters.

We first briefly review some concepts from Inductive Logic Programming and then discuss systems for Probabilistic Inductive Logic Programming (PILP) [De Raedt et al., 2008; Riguzzi et al., 2014].

10.1 Inductive Logic Programming

The ILP field is concerned with learning logic programs from data. One of the learning problems that is considered is *learning from entailment*.

Definition 52 (Inductive Logic Programming – learning from entailment). *Given two sets $E^+ = \{e_1, \dots, e_T\}$ and $E^- = \{e_{T+1}, \dots, e_Q\}$ of ground atoms (positive and negative examples), a logic program B (background knowledge), and a space of possible programs \mathcal{H} (language bias), find a program $P \in \mathcal{H}$ such that*

- $\forall e \in E^+, P \cup B \models e$ (completeness),
- $\forall e \in E^-, P \cup B \not\models e$ (consistency),

An example e such that $P \cup B \models e$ is said to be covered. We also define the following functions

- $\text{covers}(P, e) = \text{true}$ if $B \cup P \models e$,
- $\text{covers}(P, E) = \{e \in E \mid \text{covers}(P, e) = \text{true}\}$.

The predicates for which we are given examples are called target predicates. There is often a single target predicate.

Example 100 (ILP problem). *Suppose we have*

$$\begin{aligned}
 E^+ &= \{ \text{father}(\text{john}, \text{mary}), \text{father}(\text{david}, \text{steve}) \} \\
 E^- &= \{ \text{father}(\text{kathy}, \text{mary}), \text{father}(\text{john}, \text{steve}) \} \\
 B &= \text{parent}(\text{john}, \text{mary}), \\
 &\quad \text{parent}(\text{david}, \text{steve}), \\
 &\quad \text{parent}(\text{kathy}, \text{mary}), \\
 &\quad \text{female}(\text{kathy}), \\
 &\quad \text{male}(\text{john}), \\
 &\quad \text{male}(\text{david}) \}
 \end{aligned}$$

then a solution of the learning from entailment problem is

$$\text{father}(X, Y) \leftarrow \text{parent}(X, Y), \text{male}(X).$$

ILP systems that solve the learning from entailment problem differ in the way the language bias is expressed and in the strategy for searching the program space. They are usually based on two nested loops, an external *covering loop* which adds a clause to the current theory and removes the covered positive examples, and an internal *clause search loop* that searches the space of clause. Examples of ILP systems are FOIL [Quinlan, 1990], mFOIL [Dzeroski, 1993], Aleph [Srinivasan, 2007], and Progol [Muggleton, 1995].

The space of clauses is organized in terms of a generality relation that directs the search. A clause C is *more general* than D if $\text{covers}(\{C\}, U) \supseteq \text{covers}(\{D\}, U)$ where U is the set of all ground atoms built over target predicates. If $B, \{C\} \models D$, then C is more general than D because $B, \{D\} \models e$ implies $B, \{C\} \models e$. However, entailment is semi-decidable, so simpler generality relations are used in practice. The one most used is θ -subsumption: C θ -subsumes D ($C \geq D$) if there exists a substitution θ such that $C\theta \subseteq D$ [Plotkin, 1970], where the clauses are intended as sets of literals. If $C \geq D$, then $C \models D$, so C is more general than D . However, the opposite is not true in general, i.e., $C \geq D \not\Rightarrow C \models D$. While θ -subsumption is not equivalent to entailment, it is decidable (even if NP-complete), so it is chosen as the generality relation in practice.

Example 101 (Examples of theta subsumption). *Let*

$$C_1 = \text{father}(X, Y) \leftarrow \text{parent}(X, Y)$$

$$C_2 = \text{father}(X, Y) \leftarrow \text{parent}(X, Y), \text{male}(X)$$

$$C_3 = \text{father}(\text{john}, \text{steve}) \leftarrow \text{parent}(\text{john}, \text{steve}), \text{male}(\text{john})$$

Then

- $C_1 \geq C_2$ with $\theta = \emptyset$;
- $C_1 \geq C_3$ with $\theta = \{X/\text{john}, Y/\text{steve}\}$;
- $C_2 \geq C_3$ with $\theta = \{X/\text{john}, Y/\text{steve}\}$.

ILP systems differ in the direction of search in the space of clauses ordered by generality: top-down systems search the space from more general clauses to less general ones and bottom-up systems do the opposite. Aleph and Progol are examples of top-down systems. In top-down systems, the clause search loop consists of gradually specializing clauses using heuristics to guide the search, for example, by using *beam search*. Clause specializations are obtained by applying a *refinement operator* ρ that, given a clause C , returns a set of its specializations, i.e., $\rho(C) \subseteq \{D \mid D \in \mathcal{L}, C \geq D\}$ where \mathcal{L} is the space of possible clauses. A refinement operator usually generates only minimal specializations and typically applies two syntactic operations:

- a substitution, or
- the addition of a literal to the body.

In Progol, for example, the refinement operator adds a literal from the *bottom clause* \perp after having replaced some of the constants with variables. \perp is the most specific clause covering an example e , i.e., $\perp = e \leftarrow B_e$, where B_e is set of ground literals that are true regarding example e that are allowed by the language bias. In this way, we are sure that, at all times during the search, the refinements at least cover example e , that can be selected at random from the set of positive examples.

In turn, the language bias is expressed in Progol by means of *mode declarations*. Following [Muggleton, 1995], a mode declaration m is either a head declaration $\text{modeh}(r, s)$ or a body declaration $\text{modeb}(r, s)$, where s , the *schema*, is a ground literal and r is an integer called the *recall*. A schema is a template for literals in the head or body of a clause and can contain special placemaker terms of the form $\#type$, $+type$, and $-type$, which stand, respectively, for ground terms, input variables, and output variables of a type. An input variable in a body literal of a clause must be either an input variable in the head or an output variable in a preceding body literal

in the clause. If M is a set of mode declarations, $L(M)$ is the *language of* M , i.e., the set of clauses $\{C = h \leftarrow b_1, \dots, b_m\}$ such that the head atom h (resp. body literals b_i) is obtained from some head (resp. body) declaration in M by replacing all $\#$ placemarkers with ground terms and all $+$ (resp. $-$) placemarkers with input (resp. output) variables.

The bottom clause is built with a procedure called *saturation*, shown in Algorithm 25. This method is a deductive procedure used to find atoms related to e . Suppose $modeh(r, s)$ is a head declaration such that e is an answer for the goal $schema(s)$, where $schema(s)$ denotes the literal obtained from s by replacing all placemarkers with distinct variables X_1, \dots, X_n .

The terms in e are used to initialize a growing set of input terms $InTerms$: these are the terms corresponding to $+$ placemarkers in s . Then each body declaration m is considered in turn. The terms from $InTerms$ are substituted into the $+$ placemarkers of m to generate a set Q of goals. Each goal is then executed against the database and up to r (the recall) successful ground instances (or all if $r = \star$) are added to the body of the clause. Any term corresponding to a $-$ placemaker in m is inserted in $InTerms$ if it is not already present. This cycle is repeated for an user-defined number NS of times.

The resulting ground clause $\perp = e \leftarrow b_1, \dots, b_m$ is then processed to obtain a program clause by replacing each term in a $+$ or $-$ placemaker with a variable, using the same variable for identical terms. Terms corresponding to $\#$ placemarkers are instead kept in the clause.

Example 102 (Bottom clause example). *Consider the learning problem of Example 100 and the language bias*

$modeh(father(+person, +person)).$
 $modeb(parent(+person, -person)).$
 $modeb(parent(\#person, +person)).$
 $modeb(male(+person)).$
 $modeb(female(\#person)).$

then the bottom clause for $father(john, mary)$ is

$father(john, mary) \leftarrow parent(john, mary), male(john),$
 $parent(kathy, mary), female(kathy).$

After replacing constants with variables we get

$father(X, Y) \leftarrow parent(X, Y), male(X), parent(kathy, Y),$
 $female(kathy).$

Algorithm 25 Function SATURATION.

```

1: function SATURATION( $e, r, NS$ )
2:    $InTerms = \emptyset$ ,
3:    $\perp = \emptyset$  ▷  $\perp$ : bottom clause
4:   for all arguments  $t$  of  $e$  do
5:     if  $t$  corresponds to a  $+type$  then
6:       add  $t$  to  $InTerms$ 
7:     end if
8:   end for
9:   let  $\perp$ 's head be  $e$ 
10:  repeat
11:     $Steps \leftarrow 1$ 
12:    for all modeb declarations  $modeb(r, s)$  do
13:      for all possible subs.  $\sigma$  of variables corresponding to  $+type$  in  $schema(s)$  by terms from
       $InTerms$  do
14:        for  $j = 1 \rightarrow r$  do
15:          if goal  $b = schema(s)$  succeeds with answer substitution  $\sigma'$  then
16:            for all  $v/t \in \sigma$  and  $\sigma'$  do
17:              if  $v$  corresponds to a  $-type$  then
18:                add  $t$  to the set  $InTerms$  if not already present
19:              end if
20:            end for
21:          Add  $b$  to  $\perp$ 's body
22:        end if
23:      end for
24:    end for
25:  end for
26:   $Steps \leftarrow Steps + 1$ 
27:  until  $Steps > NS$ 
28:  replace constants with variables in  $\perp$ , using the same variable for identical terms
29:  return  $\perp$ 
30: end function

```

10.2 LLPAD and ALLPAD Structure Learning

LLPAD [Riguzzi, 2004] and ALLPAD [Riguzzi, 2007b, 2008b] were two early systems for performing structure learning. They learned ground LPADs from interpretations. We discussed parameter learning in Section 9.2; we consider here the problem of learning the structure.

Definition 53 (ALLPAD Structure learning problem). *Given a set*

$$E = \{(I, p_I) \mid I \in Int2, p_I \in [0, 1]\}$$

such that $\sum_{(I,p_I) \in E} p_I = 1$, and a space of possible programs \mathcal{S} , find an LPAD $\mathcal{P} \in \mathcal{S}$ such that

$$Err = \sum_{(I,p_I) \in E} |P(I) - p_I|$$

is minimized, where $P(I)$ is the probability assigned by \mathcal{P} to I .

As for parameter learning, E may also be given as a multiset E' of interpretations.

LLPAD and ALLPAD learn ground programs satisfying the exclusive-or assumption, so each couple of clauses that share an atom in the head has mutually exclusive bodies, i.e., not both true in the same interpretation from \mathcal{I} .

The systems perform learning in three phases: they first find a set of clause structures satisfying some constraints, then compute the annotations of head atoms of such clauses using Theorem 14, and finally solve a constraint optimization problem for selecting a subset of clauses to include in the solution.

The first phase can be cast in the framework proposed by [Stolle et al., 2005] in which the problem of descriptive ILP is seen as the problem of finding all the clauses in the language bias that satisfy a number of constraints. Exploiting the properties of constraints, the search in the space of clauses can be usefully pruned.

A constraint is *monotonic* if it is the case that when a clause does not satisfy it, none of its generalizations (in the θ -subsumption generalization order) satisfy it. A constraint is *anti-monotonic* if it is the case that when a clause does not satisfy it, none of its specializations satisfy it.

The first phase can be formulated in this way: find all the disjunctive clauses that satisfy the following constraints:

- C1 have their body true in at least one interpretation;
- C2 are satisfied in all the interpretations;
- C3 their atoms in the head are mutually exclusive over the set of interpretations where the body is true (i.e., no two head atoms are both true in an interpretation where the body is true);
- C4 they have no redundant head atom, i.e., no head atom that is false in all the interpretations where the body is true.

The systems search the space of disjunctive clauses by first searching depth-first and top-down for bodies true in at least one interpretation and then, for

each such body, searching for a head satisfying the remaining constraints. When a body is found that is true in zero interpretations, the search along that branch is stopped (constraint C1 is anti-monotonic).

The systems employ bottom-up search in the space of heads, exploiting the monotonic constraint C2 that requires the clause to be true in all the interpretations for pruning the search.

The second phase is performed using Theorem 14: given a ground clause generated by the second phase, the probabilities of head atoms are given by the conditional probability of the head atoms given the body according to the distribution Pr .

In the third phase, the systems associate a Boolean decision variable with each of the clauses found in the first phase. Thanks to the exclusive-or assumption, the probability of each interpretation can be expressed as a function of the decision variables, so we set Err as the objective function of an optimization problem. The exclusive-or assumption is enforced by imposing constraints among couples of decisions.

Both the constraints and the objective function are linear, so we can use mixed-integer programming techniques.

The restriction to ground programs satisfying the exclusive-or assumption limits the applicability of LLPAD and ALLPAD in practice.

10.3 ProbLog Theory Compression

De Raedt et al. [2008] consider the problem of compressing a ProbLog theory given a set of positive and negative examples. The problem can be defined as follows.

Definition 54 (Theory compression). *Given a ProbLog program \mathcal{P} containing the set of probabilistic facts $\mathcal{F} = \{\Pi_1 :: f_1, \dots, \Pi_n :: f_n\}$, two sets $E^+ = \{e_1, \dots, e_T\}$ and $E^- = \{e_{T+1}, \dots, e_Q\}$ of ground atoms (positive and negative examples), and a constant $k \in \mathbb{N}$, find a subset of the probabilistic facts $\mathcal{G} \subseteq \mathcal{F}$ of size at most k (i.e., $|\mathcal{G}| \leq k$) that maximizes the likelihood of the examples, i.e., solve*

$$\arg \max_{\mathcal{G} \subseteq \mathcal{F}, |\mathcal{G}| \leq k} \prod_{i=1}^T P(e_i) \prod_{i=T+1}^Q P(\sim e_i)$$

The aim is to modify the theory by removing clauses in order to maximize the likelihood of the examples. This is an instance of a *theory revision*

process. However, in case an example has probability 0, the whole likelihood would be 0. To be able to consider also this case, $P(e)$ is replaced by $\hat{P}(e) = \min(\epsilon, P(e))$ for a small user-defined constant ϵ .

The ProbLog compression algorithm of [De Raedt et al., 2008] proceeds by greedily removing one probabilistic fact at a time from the theory. The fact is chosen as the one whose removal results in the largest likelihood increase. The algorithm continues removing facts if there are more than k of them and there is a fact whose removal can improve the likelihood.

The algorithm first builds the BDDs for all the examples. Then it enters the removal cycle. Computing the effect of the removal of a probabilistic fact f_i on the probability of an example is easy: it is enough to set Π_i to 0 and re-evaluate the BDD using function PROB of Algorithm 4. The value of the root will be the updated probability of the example. Computing the likelihood after the removal of a probabilistic fact is thus quick, as the expensive construction of the BDDs does not have to be redone.

10.4 ProbFOIL and ProbFOIL+

ProbFOIL [De Raedt and Thon, 2011] and ProbFOIL+ [Raedt et al., 2015] learn rules from probabilistic examples. The learning problem they consider is defined as follows.

Definition 55 (ProbFOIL/ProbFOIL+ learning problem [Raedt et al., 2015]).
Given

1. a set of training examples $E = \{(e_1, p_1), \dots, (e_T, p_T)\}$ where each e_i is a ground fact for a target predicate;
2. a background theory \mathcal{B} containing information about the examples in the form of a ProbLog program;
3. a space of possible clauses \mathcal{L} .

find a hypothesis $H \subseteq \mathcal{L}$ so that the absolute error $AE = \sum_{i=1}^T |P(e_i) - p_i|$ is minimized, i.e.,

$$\arg \min_{H \in \mathcal{L}} \sum_{i=1}^T |P(e_i) - p_i|$$

The difference between ProbFOIL and ProbFOIL+ is that in ProbFOIL the clauses in H are definite, i.e., of the form $h \leftarrow B$, while in ProbFOIL+

they are probabilistic, i.e., of the form $x :: h \leftarrow B$, with $x \in [0, 1]$. Such rules are to be interpreted as the combination of

$$\begin{aligned} h &\leftarrow B, \text{prob}(id). \\ x &:: \text{prob}(id). \end{aligned}$$

where id is an identifier of the rule and $x :: \text{prob}(id)$ is a ground probabilistic fact associated with the rule. Note that this is different from an LPAD rule of the form $h : x \leftarrow B$, as this stands for the union of ground rules $h' : x \leftarrow B'$ obtained by grounding $h : x \leftarrow B$. So LPAD rules generate an independent random variable for each of their groundings, while the above ProbFOIL+ rule generates a single random variable independently on the number of groundings.

ProbFOIL+ generalizes the mFOIL system [Dzeroski, 1993], itself a generalization of FOIL [Quinlan, 1990]. It adopts the standard technique for learning sets of rules consisting of a covering loop in which one rule is added to the theory at each iteration. A nested clause search loop builds the rule by iteratively adding literals to the body of the rule. The covering loop ends when a condition based on a global scoring function is satisfied. The construction of single rules is performed by means of beam search as in mFOIL and uses a local scoring function as the heuristic. Algorithm 26 shows the overall approach¹.

The global scoring function is the accuracy over the dataset, given by

$$\text{accuracy}_H = \frac{TP_H + TN_H}{T}$$

where T is the number of examples and TP_H and TN_H are, respectively, the number of *true positives* and of *true negatives*, i.e., the number of positive (negative) examples correctly classified as positive (negative).

The local scoring function is an *m-estimate* [Mitchell, 1997] of the *precision*, or the probability that an example is positive given that it is covered by the rule:

$$m\text{-estimate}_H = \frac{TP_H + m \frac{P}{P+N}}{TP_H + FP_H + m}$$

where m is a parameter of the algorithm, FP_H is the number of *false positives* (negative examples classified as positive), and P and N indicate the number of positive and negative examples in the dataset, respectively.

These measures are based on usual metrics for rule learning that assume that the training set is composed of positive and negative examples and that

¹The description of ProbFOIL+ is based on [Raedt et al., 2015] and the code at <https://bitbucket.org/antondries/prob2foil>

Algorithm 26 Function **PROBFOIL+**.

```

1: function PROBFOIL+(target)
2:    $H \leftarrow \emptyset$ 
3:   while true do
4:      $clause \leftarrow \text{LEARNRULE}(H, target)$ 
5:     if GSCORE( $H$ ) < GSCORE( $H \cup \{clause\}$ )  $\wedge$  SIGNIFICANT( $H, clause$ ) then
6:        $H \leftarrow H \cup \{clause\}$ 
7:     else
8:       return  $H$ 
9:     end if
10:  end while
11: end function
12: function LEARNRULE( $H, target$ )
13:   $candidates \leftarrow \{x :: target \leftarrow true\}$ 
14:   $best \leftarrow (x :: target \leftarrow true)$ 
15:  while  $candidates \neq \emptyset$  do
16:     $next\_cand \leftarrow \emptyset$ 
17:    for all  $x :: target \leftarrow body \in candidates$  do
18:      for all ( $target \leftarrow bod, refinement$ )  $\in \rho(target \leftarrow body)$  do
19:        if not REJECT( $H, best, (x :: target \leftarrow body, refinement)$ ) then
20:           $next\_cand \leftarrow next\_cand \cup \{(x :: target \leftarrow body, refinement)\}$ 
21:          if LSCORE( $H, (x :: target \leftarrow body, refinement)$ ) > LSCORE( $H, best$ ) then
22:             $best \leftarrow (x :: target \leftarrow body, refinement)$ 
23:          end if
24:        end if
25:      end for
26:    end for
27:     $candidates \leftarrow next\_cand$ 
28:  end while
29:  return  $best$ 
30: end function

```

classification is sharp. ProbFOIL+ generalizes this settings as each example e_i is associated with a probability p_i . The deterministic setting is obtained by having $p_i = 1$ for positive examples and $p_i = 0$ for negative examples. In the probabilistic setting, we can see an example (e_i, p_i) as contributing a part p_i to the positive part of training set and $1 - p_i$ to the negative part. So in this case, $P = \sum_{i=1}^T p_i$ and $N = \sum_{i=1}^T (1 - p_i)$. Similarly, prediction is generalized: the hypothesis H assigns a probability $p_{H,i}$ to each example e_i , instead of simply saying that the example is positive ($p_{H,i} = 1$) or negative ($p_{H,i} = 0$). The number of true positive and true negatives can be generalized as well. The contribution $tp_{H,i}$ of example e_i to TP_H will be $p_{H,i}$ if $p_i > p_{H,i}$ and p_i otherwise, because if $p_i < p_{H,i}$, the hypothesis is overestimating e_i . The contribution $fp_{H,i}$ of example e_i to FP_H will be $p_{H,i} - p_i$ if $p_i < p_{H,i}$ and 0 otherwise, because if $p_i > p_{H,i}$ the hypothesis is underestimating e_i . Then $TP_H = \sum_{i=1}^T tp_{H,i}$, $FP_H = \sum_{i=1}^T fp_{H,i}$, $TN_H = N - FP_H$, and

$FN_H = P - TP_H$ as for the deterministic case, where FN_H is the number of false negatives, or positive examples classified as negatives.

The function $LSCORE(H, x :: C)$ computes the local scoring function for the addition of clause $C(x) = x :: C$ to H using the m-estimate. However, the heuristic depends on the value of $x \in [0, 1]$. Thus, the function has to find the value of x that maximizes the score

$$M(x) = \frac{TP_{H \cup C(x)} + mP/T}{TP_{H \cup C(x)} + FP_{H \cup C(x)} + m}.$$

To do so, we need to compute $TP_{H \cup C(x)}$ and $FP_{H \cup C(x)}$ as a function of x . In turn, this requires the computation of $tp_{H \cup C(x),i}$ and $fp_{H \cup C(x),i}$, the contributions of each example.

Note that $p_{H \cup C(x),i}$ is monotonically increasing in x , so the minimum and maximum values are obtained for $x = 0$ and $x = 1$, respectively. Let us call them l_i and u_i , so $l_i = p_{H \cup C(0),i} = p_{H,i}$ and $u_i = p_{H \cup C(1),i}$. Since ProbFOIL differs from ProbFOIL+ only for the use of deterministic clauses instead of probabilistic ones, u_i is the value that is used by ProbFOIL for the computation of $LSCORE(H, C)$ which thus returns $M(1)$.

In ProbFOIL+, we need to study the dependency of $p_{H \cup C(x),i}$ on x . If the clause is deterministic, it adds probability mass $u_i - l_i$ to $p_{H,i}$. We can imagine the u_i as being the probability that the Boolean formula $F = X_H \vee \neg X_H \wedge X_B$ takes value 1, with X_H a Boolean variable that is true if H covers the example, $P(X_H) = p_{H,i}$, X_B a Boolean variable that is true if the body of clause C covers the example and $P(\neg X_H \wedge X_B) = u_i - l_i$. In fact, since the two Boolean terms are mutually exclusive, $P(F) = P(X_H) + P(\neg X_H \wedge X_B) = p_{H,i} + u_i - p_{H,i} = u_i$. If the clause is probabilistic, its random variable X_C is independent from all the other random variables, so the probability of the example can be computed as the probability that the Boolean function $F' = X_H \vee X_C \wedge \neg X_H \wedge X_B$ takes value 1, with $P(X_C) = x$. So $p_{H \cup C(x),i} = P(F') = p_{H,i} + x(u_i - l_i)$ and $p_{H \cup C(x),i}$ is a linear function of x .

We can isolate the contribution of $C(x)$ to $tp_{H \cup C(x),i}$ and $fp_{H \cup C(x),i}$ as follows:

$$tp_{H \cup C(x),i} = tp_{H,i} + tp_{C(x),i} \quad fp_{H \cup C(x),i} = fp_{H,i} + fp_{C(x),i}$$

Then the examples can be grouped into three sets:

E_1 : $p_i \leq l_i$, the clause overestimates the example independently of the value of x , so $tp_{C(x),i} = 0$ and $fp_{C(x),i} = x(u_i - l_i)$

E_2 : $p_i \geq u_i$, the clause underestimates the example independently of the value of x , so $tp_{C(x),i} = x(u_i - l_i)$ and $fp_{C(x),i} = 0$

E_3 : $l_i \leq p_i \leq u_i$, there is a value of x for which the clause predicts the correct probability for the example. This value is obtained by solving $x(u_i - l_i) = p_i - l_i$ for x , so

$$x_i = \frac{p_i - l_i}{u_i - l_i}$$

For $x \leq x_i$, $tp_{C(x),i} = x(u_i - l_i)$ and $fp_{C(x),i} = 0$. For $x > x_i$, $tp_{C(x),i} = p_i - l_i$ and $fp_{C(x),i} = x(u_i - l_i) - (p_i - l_i)$.

We can express $TP_{H \cup C(x)}$ and $FP_{H \cup C(x)}$ as

$$\begin{aligned} TP_{H \cup C(x)} &= TP_H + TP_1(x) + TP_2(x) + TP_3(x) \\ FP_{H \cup C(x)} &= FP_H + FP_1(x) + FP_2(x) + FP_3(x) \end{aligned}$$

where $TP_l(x)$ and $FP_l(x)$ are the contribution of the set of examples E_l . These can be computed as

$$TP_1(x) = 0$$

$$FP_1(x) = x \sum_{i \in E_1} (u_i - l_i) = xU_1$$

$$TP_2(x) = x \sum_{i \in E_2} (u_i - l_i) = xU_2$$

$$FP_2(x) = 0$$

$$TP_3(x) = x \sum_{i: i \in E_3, x \leq x_i} (u_i - l_i) + \sum_{i: i \in E_3, x > x_i} (p_i - l_i) = xU_3^{\leq x_i} + P_3^{> x_i}$$

$$FP_3(x) = x \sum_{i: i \in E_3, x > x_i} (u_i - l_i) - \sum_{i: i \in E_3, x > x_i} (p_i - l_i) = xU_3^{> x_i} - P_3^{> x_i}$$

By replacing these formulas into $M(x)$, we get

$$M(x) = \frac{(U_2 + U_3^{\leq x_i})x + TP_H + P_3^{> x_i} + mP/T}{(U_1 + U_2 + U_3)x + TP_H + FP_H + m}$$

where $U_3 = x \sum_{i \in E_3} (u_i - l_i) = (TP_3(x) + FP_3(x))/x$.

Since $U_3^{\leq x_i}$ and $P_3^{> x_i}$ are constant in the interval between two consecutive values of x_i , $M(x)$ is a piecewise function where each piece is of the form

$$\frac{Ax + B}{Cx + D}$$

with A, B, C , and D constants. The derivative of a piece is

$$\frac{dM(x)}{dx} = \frac{AD - BC}{(Cx + D)^2}$$

which is either 0 or different from 0 everywhere in each interval, so the maximum of $M(x)$ can occur only at the x_i s values that are the endpoints of the intervals. Therefore, we can compute the value of $M(x)$ for each x_i and pick the maximum. This can be done efficiently by ordering the x_i values and computing $U_3^{\leq x_i} = \sum_{i:i \in E_3, x \leq x_i} (u_i - l_i)$ and $P_3^{> x_i} = \sum_{i:i \in E_3, x > x_i} (p_i - l_i)$ for increasing values of x_i , incrementally updating $U_3^{\leq x_i}$ and $P_3^{> x_i}$.

ProbFOIL+ prunes refinements (line 19 of Algorithm 26) when they cannot lead to a local score higher than the current best, when they cannot lead to a global score higher than the current best or when they are not significant, i.e., when they provide only a limited contribution.

By adding a literal to a clause, the true positives and false positives can only decrease, so we can obtain an upper bound of the local score that any refinement can achieve by setting the false positives to 0 and computing the m-estimate. If this value is smaller than the current best, the refinement is discarded.

By adding a clause to a theory, the true positives and false positives can only increase, so if the number of true positives of $H \cup C(x)$ is not larger than the true positives of H , the refinement $C(x)$ can be discarded.

ProbFOIL+ performs a *significance test* borrowed from mFOIL that is based on the *likelihood ratio statistics*.

ProbFOIL+ computes a statistics $LhR(H, C)$ that takes into account the effect of the addition of C to H on TP and FP so that a clause is discarded if it has limited effect. $LhR(H, C)$ is distributed according to χ^2 with one degree of freedom, so the clause can be discarded if $LhR(H, C)$ is outside the interval for the confidence chosen by the user.

Another system that solves the ProbFOIL learning problem is SKILL [Côte-Real et al., 2015]. Differently from ProbFOIL, it is based on the ILP system TopLog [Muggleton et al., 2008]. In order to prune the universe of candidate theories and speed up learning, SKILL uses estimates of the predictions of theories [Côte-Real et al., 2017].

10.5 SLIPCOVER

SLIPCOVER [Bellodi and Riguzzi, 2015] learns LPADs by first identifying good candidate clauses and then by searching for a theory guided by the LL of the data. As EMBLEM (see Section 9.4), it takes as input a set of mega-examples and an indication of which predicates are target, i.e., those for which we want to optimize the predictions of the final theory. The mega-examples must contain positive and negative examples for all predicates that may appear in the head of clauses, either target or non-target (background predicates).

10.5.1 The Language Bias

The language bias for clauses is expressed by means of *mode* declarations, as in Progol [Muggleton, 1995], see Section 10.1. SLIPCOVER extends this type of mode declarations with placemaker terms of the form $- \#$ which are treated as $\#$ when variabilizing the clause and as $-$ when performing saturation, see Section 10.5.2.1.

SLIPCOVER also allows head declarations of the form

$$\text{modeh}(r, [s_1, \dots, s_n], [a_1, \dots, a_n], [P_1/Ar_1, \dots, P_k/Ar_k]).$$

These are used to generate clauses with more than two head atoms: s_1, \dots, s_n are schemas, a_1, \dots, a_n are atoms such that a_i is obtained from s_i by replacing placemarkers with variables, and P_i/Ar_i are the predicates admitted in the body. a_1, \dots, a_n are used to indicate which variables should be shared by the atoms in the head.

Examples of mode declarations can be found in Section 10.5.3.

10.5.2 Description of the Algorithm

The main function is shown by Algorithm 27: after the search in the space of clauses, encoded in lines 2–27, SLIPCOVER performs a greedy search in the space of theories, described in lines 28–38.

The first phase aims at finding a set of promising ones (in terms of LL of the data), that will be used in the following greedy search phase. By starting from promising clauses, the greedy search is able to generate good final theories. The search in the space of clauses is split in turn in two steps: (1) the construction of a set of beams containing bottom clauses (function INITIALBEAMS at line 2 of Algorithm 27) and (2) a beam search over each of these beams to refine the bottom clauses (function CLAUSEREFINEMENTS

at line 11). The overall output of this search phase is represented by two lists of promising clauses: TC for target predicates and BC for background predicates. The clauses found are inserted either in TC , if a target predicate appears in their head, or in BC . These lists are sorted by decreasing LL.

Algorithm 27 Function SLIPCOVER.

```

1: function SLIPCOVER( $NInt, NS, NA, NI, NV, NB, NTC, NBC, D, NEM, \epsilon, \delta$ )
2:    $IB = \text{INITIALBEAMS}(NInt, NS, NA)$  ▷ Clause search
3:    $TC \leftarrow []$ 
4:    $BC \leftarrow []$ 
5:   for all ( $PredSpec, Beam$ )  $\in IB$  do
6:      $Steps \leftarrow 1$ 
7:      $NewBeam \leftarrow []$ 
8:     repeat
9:       while  $Beam$  is not empty do
10:        remove the first triple ( $Cl, Literals, LL$ ) from  $Beam$  ▷ Remove the first clause
11:         $Refs \leftarrow \text{CLAUSEREFINEMENTS}((Cl, Literals), NV)$  ▷ Find all refinements  $Refs$ 
        of ( $Cl, Literals$ ) with at most  $NV$  variables
12:        for all ( $Cl', Literals'$ )  $\in Refs$  do
13:          ( $LL'', \{Cl''\}$ )  $\leftarrow \text{EMBLEM}(\{Cl'\}, D, NEM, \epsilon, \delta)$ 
14:           $NewBeam \leftarrow \text{INSERT}((Cl'', Literals', LL''), NewBeam, NB)$ 
15:          if  $Cl''$  is range-restricted then
16:            if  $Cl''$  has a target predicate in the head then
17:               $TC \leftarrow \text{INSERT}((Cl'', LL''), TC, NTC)$ 
18:            else
19:               $BC \leftarrow \text{INSERT}((Cl'', LL''), BC, NBC)$ 
20:            end if
21:          end if
22:        end for
23:      end while
24:       $Beam \leftarrow NewBeam$ 
25:       $Steps \leftarrow Steps + 1$ 
26:    until  $Steps > NI$ 
27:  end for
28:   $Th \leftarrow \emptyset, ThLL \leftarrow -\infty$  ▷ Theory search
29:  repeat
30:    remove the first couple ( $Cl, LL$ ) from  $TC$ 
31:    ( $LL', Th'$ )  $\leftarrow \text{EMBLEM}(Th \cup \{Cl\}, D, NEM, \epsilon, \delta)$ 
32:    if  $LL' > ThLL$  then
33:       $Th \leftarrow Th', ThLL \leftarrow LL'$ 
34:    end if
35:  until  $TC$  is empty
36:   $Th \leftarrow Th \cup_{(Cl, LL) \in BC} \{Cl\}$ 
37:  ( $LL, Th$ )  $\leftarrow \text{EMBLEM}(Th, D, NEM, \epsilon, \delta)$ 
38:  return  $Th$ 
39: end function

```

The second phase starts with an empty theory Th which is assigned the lowest value of LL (line 28 of Algorithm 27). Then one target clause Cl at a

time is added from the list TC . After each addition, parameter learning with EMBLEM is run on the extended theory $Th \cup \{Cl\}$ and the LL LL' of the data is used as the score of the resulting theory Th' . If LL' is better than the current best, the clause is kept in the theory; otherwise, it is discarded (lines 31–34). This is done for each clause in TC .

Finally, SLIPCOVER adds all the (background) clauses from the list BC to the theory composed of target clauses only (line 36) and performs parameter learning on the resulting theory (line 37). The clauses that are never used to derive the examples will get a value of 0 for the parameters of the atoms in their head and will be removed in a post-processing phase.

In the following, we provide a detailed description of the two support functions for the first phase, the search in the space of clauses.

10.5.2.1 Function INITIALBEAMS

Algorithm 28 shows how the initial set of beams IB , one for each predicate P (with arity Ar) appearing in a modeh declaration, is generated by building a set of bottom clauses as in Progol, see Section 10.1. The algorithm outputs the initial clauses that will be then refined by Function CLAUSEREFINEMENTS.

In order to generate a bottom clause for a mode declaration $modeh(r, s)$ specified in the language bias, an input mega-example I is selected and an answer h for the goal $schema(s)$ is selected, where $schema(s)$ denotes the literal obtained from s by replacing all placemarkers with distinct variables X_1, \dots, X_n (lines 5–9 of Algorithm 28). The mega-example and the atom h are both randomly sampled with replacement, the former from the available set of training mega-examples and the latter from the set of all answers found for the goal $schema(s)$ in the mega-example. Each of these answers represents a positive example.

Then h is saturated using Algorithm 25 modified so that, when a term in an answer substitution (line 17) corresponds to a $-#type$ argument, it is added to $InTerms$ as for $-type$ arguments. Moreover, when replacing constants with variables, terms corresponding to $-#$ placemarkers are kept in the clause as for $\#$ placemaker. This is useful when we want to test the equality of the value of an argument with a constant but we also want to retrieve other atoms related to that constant.

The initial beam $Beam$ associated with predicate P/Ar of h contains the clause with empty body $h : 0.5 \leftarrow true$ for each bottom clause of the form $h : - b_1, \dots, b_m$ (lines 10 and 11 of Algorithm 28). This process is repeated for a number $NInt$ of input mega-examples and a number NA of answers, thus obtaining $NInt \cdot NA$ bottom clauses.

The generation of a bottom clause for a mode declaration

$$m = \text{modeh}(r, [s_1, \dots, s_n], [a_1, \dots, a_n], [P_1/Ar_1, \dots, P_k/Ar_k])$$

is the same except for the fact that the goal to call is composed of more than one atom. In order to build the head, the goal a_1, \dots, a_n is called and NA answers that ground all a_i s are kept (lines 15–19). From these, the set of input terms $InTerms$ is built and body literals are found by Function SATURATION (line 20 of Algorithm 28) as above. The resulting bottom clauses then have the form

$$a_1 ; \dots ; a_n \leftarrow b_1, \dots, b_m$$

and the initial beam $Beam$ will contain clauses with an empty body of the form

$$a_1 : \frac{1}{n+1} ; \dots ; a_n : \frac{1}{n+1} \leftarrow true.$$

Finally, the set of the beams for each predicate P is returned to Function SLIPCOVER.

Algorithm 28 Function INITIALBEAMS.

```

1: function INITIALBEAMS( $NInt, NS, NA$ )
2:    $IB \leftarrow \emptyset$ 
3:   for all predicates  $P/Ar$  do
4:      $Beam \leftarrow []$ 
5:     for all modeh declarations  $\text{modeh}(r, s)$  with  $P/Ar$  predicate of  $s$  do
6:       for  $i = 1 \rightarrow NInt$  do
7:         select randomly a mega-example  $I$ 
8:         for  $j = 1 \rightarrow NA$  do
9:           select randomly an atom  $h$  from  $I$  matching  $\text{schema}(s)$ 
10:          bottom clause  $BC \leftarrow \text{SATURATION}(h, r, NS)$ , let  $BC$  be  $Head : - Body$ 
11:           $Beam \leftarrow [(h : 0.5 \leftarrow true, Body, -\infty) | Beam]$ 
12:        end for
13:      end for
14:    end for
15:    for all modeh declarations  $\text{modeh}(r, [s_1, \dots, s_n], [a_1, \dots, a_n], PL)$  with  $P/Ar$  in  $PL$ 
    appearing in  $s_1, \dots, s_n$  do
16:      for  $i = 1 \rightarrow NInt$  do
17:        select randomly a mega-example  $I$ 
18:        for  $j = 1 \rightarrow NA$  do
19:          select randomly a set of atoms  $h_1, \dots, h_n$  from  $I$  matching  $a_1, \dots, a_n$ 
20:          bottom clause  $BC \leftarrow \text{SATURATION}((h_1, \dots, h_n), r, NS)$ , let  $BC$  be
     $Head : - Body$ 
21:           $Beam \leftarrow [(a_1 : \frac{1}{n+1} ; \dots ; a_n : \frac{1}{n+1} \leftarrow true, Body, -\infty) | Beam]$ 
22:        end for
23:      end for
24:    end for
25:     $IB \leftarrow IB \cup \{(P/Ar, Beam)\}$ 
26:  end for
27:  return  $IB$ 
28: end function

```

10.5.2.2 Beam Search with Clause Refinements

SLIPCOVER then performs a cycle over each predicate, either target or background (line 5 of Algorithm 27): in each iteration, it runs a beam search in the space of clauses for the predicate (line 9).

For each clause Cl in the beam, with $Literals$ admissible in the body, Function CLAUSEREFINEMENTS, shown in Algorithm 29, computes refinements by adding a literal from $Literals$ to the body or by deleting an atom from the head in the case of multiple-head clauses with a number of disjuncts (including the *null* atom) greater than 2. Furthermore, the refinements must respect the input–output modes of the bias declarations, must be connected (i.e., each body literal must share a variable with the head or a previous body literal), and their number of variables must not exceed a user-defined number NV . The couple $(Cl', Literals')$ indicates a refined clause Cl' together with the new set $Literals'$ of literals allowed in the body of Cl' ; the tuple $(Cl'_h, Literals)$ indicates a specialized clause Cl' where one disjunct in its head has been removed.

At line 13 of Algorithm 27, parameter learning is performed using EMBLEM, see Section 9.4, on a theory composed of the single refined clause.

This clause is then inserted into a list of promising clauses: either into TC , if a target predicate appears in its head, or into BC . The insertion is in order of decreasing LL. If the clause is not range-restricted, i.e., if some of the variables in the head do not appear in a positive literal in the body, then it is not inserted in TC nor in BC . These lists have a maximum size: if an insertion increases the size over the maximum, the last element is removed. In Algorithm 27, Function INSERT(I , $Score$, $List$, N) is used to insert in order a clause I with score $Score$ in a $List$ with at most N elements. Beam search is repeated until the beam becomes empty or a maximum number NI of iterations is reached.

The separate search for clauses has similarity with the covering loop of ILP systems such as Aleph and Progol. Differently from ILP, however, the test of an example requires the computation of all its explanations, while, in ILP, the search stops at the first successful derivation. The only interaction among clauses in PLP happens if the clauses are recursive. If not, then adding clauses to a theory only adds explanations for the example – increasing its probability – so clauses can be added individually to the theory. If the clauses are recursive, the examples for the head predicates are used to resolve literals in the body; thus, the test of examples on individual clauses approximates the test on a complete theory.

Algorithm 29 Function CLAUSEREFINEMENTS.

```

1: function CLAUSEREFINEMENTS( $(Cl, Literals), NV$ )
2:    $Refs = \emptyset, Nvar = 0;$   $\triangleright Nvar$ : number of different variables in a clause
3:   for all  $b \in Literals$  do
4:      $Literals' \leftarrow Literals \setminus \{b\}$ 
5:     add  $b$  to  $Cl$  body obtaining  $Cl'$ 
6:      $Nvar \leftarrow$  number of  $Cl'$  variables
7:     if  $Cl'$  is connected  $\wedge Nvar < NV$  then
8:        $Refs \leftarrow Refs \cup \{(Cl', Literals')\}$ 
9:     end if
10:  end for
11:  if  $Cl$  is a multiple-head clause then  $\triangleright$  It has 3 or more disjuncts including the null atom
12:    remove one atom from  $Cl$  head obtaining  $Cl'_h$   $\triangleright$  Not the null atom
13:    adjust the probabilities on the remaining head atoms
14:     $Refs \leftarrow Refs \cup \{(Cl'_h, Literals')\}$ 
15:  end if
16:  return  $Refs$ 
17: end function

```

10.5.3 Execution Example

We now show an example of execution on the UW-CSE dataset [Kok and Domingos, 2005] that describes the Computer Science Department of the University of Washington with 22 different predicates, such as `advisedby/2`, `years_inprogram/2`, and `taughtby/3`. The aim is to predict the predicate `advisedby/2`, namely, the fact that a person (student) is advised by another person (professor).

The language bias contains *modeh* declarations for two-head clauses such as

```
modeh(*, advisedby(+person, +person)).
```

and *modeh* declarations for multi-head clauses such as

```
modeh(*, [advisedby(+person, +person),
tempadvisedby(+person, +person)],
[advisedby(A, B), tempadvisedby(A, B)],
[professor/1, student/1, hasposition/2, inphase/2,
publication/2,
taughtby/3, ta/3, courselevel/2, years_inprogram/2]).
```

```
modeh(*, [student(+person), professor(+person)],
[student(P), professor(P)],
[hasposition/2, inphase/2, taughtby/3, ta/3,
courselevel/2,
years_inprogram/2, advisedby/2, tempadvisedby/2]).
```

```

modeh(*, [inphase(+person, pre_qual), inphase
          (+person, post_qual),
          inphase(+person, post_generals)],
        [inphase(P, pre_qual), inphase(P, post_qual),
          inphase(P, post_generals)],
        [professor/1, student/1, taughtby/3, ta/3, courselevel/2,
          yearsinprogram/2, advisedby/2, tempadvisedby/2,
          hasposition/2]).

```

Moreover, the bias contains *modeb* declarations such as

```

modeb(*, courselevel(+course, -level)).
modeb(*, courselevel(+course, #level)).

```

An example of a two-head bottom clause that is generated from the first *modeh* declaration and the example `advisedby(person155, person101)` is

```

advisedby(A,B):0.5 :- professor(B), student(A),
                      hasposition(B,C),
                      hasposition(B, faculty), inphase(A,D), inphase
                      (A, pre_qual),
                      yearsinprogram(A,E), taughtby(F,B,G), taughtby(F,B,H),
                      taughtby(I,B,J), taughtby(I,B,J), taughtby(F,B,G),
                      taughtby(F,B,H),
                      ta(I,K,L), ta(F,M,H), ta(F,M,H), ta(I,K,L), ta(N,K,O),
                      ta(N,A,P),
                      ta(Q,A,P), ta(R,A,L), ta(S,A,T), ta(U,A,O), ta(U,A,O),
                      ta(S,A,T),
                      ta(R,A,L), ta(Q,A,P), ta(N,K,O), ta(N,A,P), ta(I,K,L),
                      ta(F,M,H).

```

An example of a multi-head bottom clause generated from the second *modeh* declaration and the examples

```

student(person218).
professor(person218).

```

is

```

student(A):0.33; professor(A):0.33 :-
  inphase(A,B),
  inphase(A, post_generals),
  yearsinprogram(A,C).

```

When searching the space of clauses for the `advisedby/2` predicate, an example of a refinement from the first bottom clause is

```
advisedby(A,B):0.5 :- professor(B).
```

EMBLEM is then applied to the theory composed of this single clause, using the positive and negative facts for `advisedby/2` as queries for which to build the BDDs. The single parameter is updated obtaining:

```
advisedby(A,B):0.108939 :- professor(B).
```

The clause is further refined to

```
advisedby(A,B):0.108939 :- professor(B),
                           hasposition(B,C).
```

An example of a refinement that is generated from the second bottom clause is

```
student(A):0.33; professor(A):0.33 :-
    inphase(A,B).
```

The updated refinement after EMBLEM is

```
student(A):0.5869; professor(A):0.09832 :-
    inphase(A,B).
```

When searching the *space of theories* for the target predicate `advisedby`, SLIPCOVER generates the program:

```
advisedby(A,B):0.1198 :- professor(B),
                           inphase(A,C).
advisedby(A,B):0.1198 :- professor(B), student(A).
```

with an LL of -350.01. After EMBLEM, we get:

```
advisedby(A,B):0.05465 :- professor(B),
                           inphase(A,C).
advisedby(A,B):0.06893 :- professor(B),
                           student(A).
```

with an LL of -318.17. Since the LL decreased, the last clause is retained and at the next iteration, a new clause is added:

```
advisedby(A,B):0.12032 :- hasposition(B,C),
                           inphase(A,D).
advisedby(A,B):0.05465 :- professor(B),
                           inphase(A,C).
advisedby(A,B):0.06893 :- professor(B), student(A).
```

10.6 Examples of Datasets

PILP systems have been applied to many datasets. Some of them are:

- UW-CSE [Kok and Domingos, 2005]: see Section 10.5.3.
- Mutagenesis [Srinivasan et al., 1996]: a classic ILP benchmark dataset for Quantitative Structure-Activity Relationship (QSAR), i.e., predicting the biological activity of chemicals from their physicochemical properties or molecular structure. In this case, the goal is to predict the mutagenicity (a property correlated with cancerogenicity) of compounds from their chemical structure.
- Carcinogenesis [Srinivasan et al., 1997]: another classic ILP benchmark dataset for QSAR where the goal is to predict the cancerogenicity of compounds from their chemical structure.
- Mondial [Schulte and Khosravi, 2012]: a dataset containing information regarding geographical regions of the world, including population size, political system, and the country border relationship.
- Hepatitis [Khosravi et al., 2012]: a dataset derived from the Discovery Challenge Workshop of ECML/PKDD 2002 containing information on laboratory examinations of hepatitis B and C infected patients. The goal is to predict the type of hepatitis of a patient.
- Bupa [McDermott and Forsyth, 2016]: diagnosing patients with liver disorders.
- NBA [Schulte and Routley, 2014]: predicting the results of basketball matches from NBA.
- Pyrimidine, Triazine [Layne and Qiu, 2005]: QSAR datasets for predicting the inhibition of dihydrofolate reductase by pyrimidines and triazines, respectively.
- Financial [Berka, 2000]: predicting the success of loan applications by clients of a bank.
- Sisyphus [Blokkeel and Struyf, 2001]: a dataset regarding clients of an insurance business, the aim is to classify households and persons in relation to private life insurance.
- Yeast [Davis et al., 2005]: predicting whether a yeast gene codes for a protein involved in metabolism.
- Event Calculus [Schwitter, 2018]: learning effect axioms for the Event Calculus [Kowalski and Sergot, 1986].

11

cplint Examples

This chapter shows some examples of programs and how the `cplint` system can be used to reason on them.

11.1 cplint Commands

`cplint` uses two Prolog modules for performing inference, `pita` for exact inference with PITA (see Section 5.6) and `mcintyre` for approximate inference with MCINTYRE (see Section 7.2). We present here the predicates provided by these two modules.

The unconditional probability of an atom can be asked using `pita` with the predicate

```
prob(+Query:atom,-Probability:float).
```

where `+` and `-` mean that the argument is input or output, respectively, and the annotation of arguments after the colon indicates their type.

The conditional probability of a query atom given an evidence atom can be asked with the predicate

```
prob(+Query:atom,+Evidence:atom,-Probability:float).
```

The BDD representing the explanations for the query atom can be obtained with the predicate

```
bdd_dot_string(+Query:atom,-BDD:string,-Var:list).
```

that returns a string encoding the BDD in the dot format of Graphviz [Koutsoufios et al., 1991]. See Section 11.3 for an example of use.

With `mcintyre`, the unconditional probability of a goal can be computed by taking a given number of samples using the predicate

```
mc_sample(+Query:atom,+Samples:int,-Probability:float).
```

Moreover, the following predicate samples arguments of queries:

```
mc_sample_arg(+Query:atom,+Samples:int,?Arg:var,-Values:list).
```

where ? means that the argument must be a variable. The predicate samples `Query` `Samples` times. `Arg` must be a variable in `Query`. The predicate returns a list of couples `L-N` in `Values` where `L` is the list of all values of `Arg` for which `Query` succeeds in a world sampled at random and `N` is the number of samples returning that list of values. If `L` is the empty list, it means that for that sample, the query failed. If `L` is a list with a single element, it means that for that sample, the query is determinate. If, in all couples `L-N`, `L` is a list with a single element, it means that the program satisfies the exclusive-or assumption.

The version

```
mc_sample_arg_first(+Query:atom,+Samples:int,?Arg:var,
                  -Values:list).
```

also samples arguments of queries but just returns the first answer of the query for each sampled world.

Conditional queries can be asked with rejection sampling or Metropolis-Hastings MCMC. In the first case, the predicate is:

```
mc_rejection_sample(+Query:atom,+Evidence:atom,
                  +Samples:int,-Successes:int,-Failures:int,
                  -Probability:float).
```

In the latter case, the predicate is

```
mc_mh_sample(+Query:atom,+Evidence:atom,Samples:int,
             +Lag:int,-Successes:int,-Failures:int,-Probability:float).
```

Moreover, the arguments of the queries can be sampled with rejection sampling and Metropolis-Hastings MCMC using

```
mc_rejection_sample_arg(+Query:atom,+Evidence:atom,
                      +Samples:int,?Arg:var,-Values:list).
mc_mh_sample_arg(+Query:atom,+Evidence:atom,
                 +Samples:int,+Lag:int,?Arg:var,-Values:list).
```

Expectations can be computed with

```
mc_expectation(+Query:atom,+Samples:int,?Arg:var,-Exp:float).
```

that returns the expected value of the argument `Arg` in `Query` computed by sampling.

The predicate

```
mc_mh_expectation(+Query:atom,+Evidence:atom,+Samples:int,
  +Lag:int,?Arg:var,-Exp:float).
```

computes conditional expectations using Metropolis-Hastings MCMC.

The `cpint` on SWISH web application [Riguzzi et al., 2016a; Alberti et al., 2017] allows the user to write and run probabilistic programs online. It is based on the SWISH [Wielemaker et al., 2015] web front-end for SWI-Prolog. `cpint` on SWISH also adds graphics capabilities to `cpint`: the results of sampling arguments can be rendered as bar charts. All the predicates shown above have a form with an extra last argument `+Options` that accepts a list of terms specifying options. If the option `bar(-Chart:dict)` is used, the predicate returns in `Chart` an SWI-Prolog dictionary to be rendered with `C3.js`¹ as a bar chart. For example, the query

```
?- mc_sample_arg(reach(s0,0,S),50,S,ValList,[bar(Chart)]).
```

from `http://cpint.eu/e/markov_chain.pl` returns a chart with a bar for each possible sampled value whose size is the number of samples returning that value.

When the program has continuous random variables, the user can build a probability density of the sampled argument. When the evidence is on ground atoms with continuous values as arguments, the user needs to use likelihood weighting or particle filtering (see Section 7.4).

The predicate

```
mc_lw_sample_arg(+Query:atom,+Evidence:atom,+Samples:int,
  ?Arg:var,-ValList:list).
```

returns in `ValList` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds and `W` is the weight computed by likelihood weighting according to `Evidence`.

In particle filtering, the evidence is a list of atoms. The predicate

```
mc_particle_sample_arg(+Query:atom,+Evidence+term,
  +Samples:int,?Arg:var,-Values:list).
```

samples the argument `Arg` of `Query` using particle filtering given `Evidence`. `Evidence` is a list of goals and `Query` can be either a single goal or a list of goals.

¹<http://c3js.org/>

The samples obtained can be used to draw the probability density function of the argument. The predicate

```
histogram(+List:list,-Chart:dict,+Options:list)
```

takes a list of weighted samples and draws a histogram of the samples using C3.js in `cplint` on SWISH.

The predicate

```
density(+List:list,-Chart:dict,+Options:list)
```

draws a line chart of the density of the weighted samples in `List`.

In `histogram/3` and `density/3`, the options can be used to specify the bounds and the number of bins on the X -axis.

The predicate

```
densities(+PriorList:list,+PostList:list,-Chart:dict,+Options:
list)
```

draws a line chart of the density of two sets of samples, usually prior and post observations. The same options as for `histogram/3` and `density/3` are recognized.

For example, the query

```
?- mc_sample_arg(val(0,X),1000,X,L0,[],),histogram(L0,Chart,[]).
```

from http://cplint.eu/e/gauss_mean_est.pl takes 1000 samples of argument `X` of `val(0,X)` and draws the density of the samples using an histogram.

For discrete arguments, the predicate

```
argbar(+Values:list,-Chart:dict)
```

returns a bar chart with a bar for each value, where `Values` is a list of couples $V-N$ with V the value and N the number of samples returning that value.

The predicates `density_r/1`, `densities_r/2`, `histogram_r/2`, and `argbar_r/1` are the counterparts of those above for drawing graphs in `cplint` on SWISH using the R language for statistical computing².

EMBLEM (see Section 9.4) can be run with the predicate

```
induce_par(+ListOfFolds:list,-Program:list)
```

that induces the parameters of a program starting from the examples contained in the indicated *folds* (groups of examples). The predicate

```
induce(+ListOfFolds:list,-Program:list)
```

instead induces a program using SLIPCOVER (see Section 10.5).

²<https://www.r-project.org/>

The induced programs can be tested on a set of folds with

```
test (+Program:list, +ListOfFolds:list, -LL:float,
     -AUCROC:float, -ROC:list, -AUCPR:float, -PR:list)
```

that returns the log likelihood of the test example (LL), the ROC and precision recall curves (ROC and PR) for rendering with C3.js, and the areas under the curves (AUCROC and AUCPR) that are standard metrics for the evaluation of machine learning algorithms [Davis and Goadrich, 2006].

Predicate `test_r/5` is similar to `test/7` but plots the graphs using R.

11.2 Natural Language Processing

In Natural Language Processing (NLP), a common task is checking whether a sentence respects a grammar. Another common task is tagging each word of a sentence with a Part-of-Speech (POS) tag. For NLP, the grammars that are used in the theory of formal languages such as context-free grammars or left corner grammars don't work well because the rules are too strict. Natural language is more flexible and is characterized by many exceptions to rules. To model natural language, probabilistic versions of the grammars above have been developed, such as Probabilistic Context-Free Grammar or Probabilistic Left Corner Grammar (PLCG). Similarly, for POS tagging, statistical tools such as HMMs give good results. These models can all be encoded with PLP [Riguzzi et al., 2017b].

11.2.1 Probabilistic Context-Free Grammars

A PCFG consists of:

1. A context-free grammar $G = (N, \Sigma, I, R)$ where N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, $I \in N$ is a distinguished start symbol, and R is a finite set of rules of the form $X \rightarrow Y_1, \dots, Y_n$, where $X \in N$ and $Y_i \in (N \cup \Sigma)$.
2. A parameter θ for each rule $\alpha \rightarrow \beta \in R$. Therefore, we have probabilistic rules of the form $\theta : \alpha \rightarrow \beta$

This kind of model can be represented by PLP. For instance, consider the PCFG

$$\begin{aligned} 0.2 : S &\rightarrow aS \\ 0.2 : S &\rightarrow bS \\ 0.3 : S &\rightarrow a \\ 0.3 : S &\rightarrow b, \end{aligned}$$

where N is $\{S\}$ and Σ is $\{a, b\}$.

The program <http://cplint.eu/e/pcfg.pl> (adapted from [Sato and Kubota, 2015]) computes the probability of strings using top-down parsing:

```
pcfg(L) :- pcfg(['S'], [], _Der, L, []).
pcfg([A|R], Der0, Der, L0, L2) :-
    rule(A, Der0, RHS),
    pcfg(RHS, [rule(A, RHS) | Der0], Der1, L0, L1),
    pcfg(R, Der1, Der, L1, L2).
pcfg([A|R], Der0, Der, [A|L1], L2) :-
    \+ rule(A, _, _),
    pcfg(R, Der0, Der, L1, L2).
pcfg([], Der, Der, L, L).
rule('S', Der, [a, 'S']) : 0.2; rule('S', Der, [b, 'S']) : 0.2;
rule('S', Der, [a]) : 0.3; rule('S', Der, [b]) : 0.3.
```

In this example, if we want to ask the probability of the string *abaa* using exact inference, we can use the query `?- prob(pcfg([a,b,a,a]), Prob)`. We obtain the value 0.0024. In this case, the grammar is not ambiguous so there exists only one derivation with probability $0.2 \cdot 0.2 \cdot 0.2 \cdot 0.3 = 0.0024$.

11.2.2 Probabilistic Left Corner Grammars

A PLCG is a probabilistic version of a *left-corner grammar* which uses the same set of rules as a PCFG. Whereas PCFGs assume top-down parsing, PLCGs are based on bottom-up parsing. PLCGs set probabilities to three elementary operations in bottom-up parsing, i.e., shift, attach and project, rather than to expansion of non-terminals. As a result, they define a class of distributions different from that of PCFGs.

Programs for PLCGs look very different from those for PCFGs. Consider the PLCG with the rules

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow a \\ S &\rightarrow b \end{aligned}$$

The program <http://cplint.eu/e/plcg.pl> (adapted from Sato et al. [2008]) below encodes such a grammar:

```
plc(Ws) :- g_call(['S'], Ws, [], [], _Der).
g_call([], L, L, Der, Der).
g_call([G|R], [G|L], L2, Der0, Der) :- % shift
    terminal(G),
```

```

g_call(R,L,L2,Der0,Der) .
g_call([G|R], [Wd|L],L2,Der0,Der) :-
  \+ terminal(G), first(G,Der0,Wd),
  lc_call(G,Wd,L,L1,[first(G,Wd)|Der0],Der1),
  g_call(R,L1,L2,Der1,Der) .
lc_call(G,B,L,L1,Der0,Der) :- % attach
  lc(G,B,Der0,rule(G, [B|RHS2])),
  attach_or_project(G,Der0,attach),
  g_call(RHS2,L,L1,[lc(G,B,rule(G, [B|RHS2])),
    attach|Der0],Der) .
lc_call(G,B,L,L2,Der0,Der) :- % project
  lc(G,B,Der0,rule(A, [B|RHS2])),
  attach_or_project(G,Der0,project),
  g_call(RHS2,L,L1,[lc(G,B,rule(A, [B|RHS2])),
    project|Der0],Der1),
  lc_call(G,A,L1,L2,Der1,Der) .
lc_call(G,B,L,L2,Der0,Der) :-
  \+ lc(G,B,Der0,rule(G, [B|_])),
  lc(G,B,Der0,rule(A, [B|RHS2])),
  g_call(RHS2,L,L1,[lc(G,B,rule(A, [B|RHS2]))|Der0],
    Der1),
  lc_call(G,A,L1,L2,Der1,Der) .
attach_or_project(A,Der,Op) :-
  lc(A,A,Der,_), attach(A,Der,Op) .
attach_or_project(A,Der,attach) :-
  \+ lc(A,A,Der,_).
lc('S','S',_Der,rule('S',['S','S'])).
lc('S',a,_Der,rule('S',[a])).
lc('S',b,_Der,rule('S',[b])).
first('S',Der,a):0.5; first('S',Der,b):0.5.
attach('S',Der,attach):0.5; attach('S',Der,project):0.5.
terminal(a). terminal(b).

```

If we want to know the probability that the string `ab` is generated by the grammar, we can use the query `?- mc_prob(plc([a,b]),P)` . and obtain ≈ 0.031 .

11.2.3 Hidden Markov Models

HMMs (see Example 65) can be used for POS tagging: words can be considered as output symbols and a sentence as the sequence of output symbols emitted by an HMM. In this case, the states are POS tags and the sequence

of states that most probably originated the sequence of output symbols is the POS tagging of the sentence. So we can perform POS tagging by solving an MPE task.

Program <http://cplint.eu/e/hmmpos.pl> (adapted from [Lager, 2018; Nivre, 2000; Sato and Kameya, 2001]) encodes a simple HMM where the output probabilities are set to 1 (for every state, there is only one possible output). The assumption is that a POS of a word depends only on the POS of the preceding word (or on the start state in case there is no preceding word). The program is:

```

hmm(O) :-hmm(_,O) .
hmm(S,O) :-
    trans(start,Q0,[]),hmm(Q0,[],S0,O),reverse(S0,S) .
hmm(Q,S0,S,[L|O]) :-
    trans(Q,Q1,S0),
    out(L,Q,S0),
    hmm(Q1,[Q|S0],S,O) .
hmm(_,S,S,[]) .
trans(start,det,_):0.30; trans(start,aux,_):0.20;
    trans(start,v,_):0.10; trans(start,n,_):0.10;
    trans(start,pron,_):0.30 .
trans(det,det,_):0.20; trans(det,aux,_):0.01;
    trans(det,v,_):0.01; trans(det,n,_):0.77;
    trans(det,pron,_):0.01 .
trans(aux,det,_):0.18; trans(aux,aux,_):0.10;
    trans(aux,v,_):0.50; trans(aux,n,_):0.01;
    trans(aux,pron,_):0.21 .
trans(v,det,_):0.36; trans(v,aux,_):0.01;
    trans(v,v,_):0.01; trans(v,n,_):0.26; trans(v,pron,_):
        :0.36 .
trans(n,det,_):0.01; trans(n,aux,_):0.25; trans(n,v,_):
        :0.39;
    trans(n,n,_):0.34; trans(n,pron,_):0.01 .
trans(pron,det,_):0.01; trans(pron,aux,_):0.45;
    trans(pron,v,_):0.52; trans(pron,n,_):0.01;
    trans(pron,pron,_):0.01 .
out(a,det,_).
out(can,aux,_).
out(can,v,_).
out(can,n,_).
out(he,pron,_).

```

For instance, we may want to know the most probable POS sequence for the sentence “he can can a can.” By using the query

```
?- mc_sample_arg( hmm(S, [he, can, can, a, can]), 100, S, O) .
```

we obtain that the sequence [pron, aux, v, det, n] appears most frequently in O .

11.3 Drawing Binary Decision Diagrams

Example 87 models the development of an epidemic or a pandemic and is <http://cplint.eu/e/epidemic.pl>:

```
epidemic:0.6; pandemic:0.3 :- flu(_), cold.
cold : 0.7.
flu(david).
flu(robert).
```

In order to compute the probability that a pandemic arises, we can call the query:

```
?- prob(pandemic, Prob) .
```

The corresponding BDD can be obtained with:

```
?- bdd_dot_string(pandemic, BDD, Var) .
```

The call returns the BDD in the form of a graph in the dot format of Graphviz that the `cplint` on SWISH system renders and visualizes as shown in Figure 11.1. Moreover, the call returns in `Var` a data structure encoding Table 11.1 that associates multivalued variable indexes with ground instantiations of rules.

The BDD built by CUDD differs from those introduced in Section 5.3 because edges to 0-children can be negated, i.e., the function encoded by the 0-child is negated before being used in the parent node. Negated edges to 0-children are represented in the graph by dotted arcs, while edges to 1-children and regular edges to 0-children with solid and dashed arcs, respectively. Moreover, the output of the BDD can be negated as well, indicated by a dotted arc connecting an Out node to the root of the diagram, as in Figure 11.1. CUDD uses this form of BDDs for computational reasons, for example, negation is very cheap, as it just requires changing the type of an edge.

Each level of the BDD is associated with a variable of the form $X_{i,k}$ indicated on the left: i indicates the multivalued variable index and k the index of the Boolean variable. The association between the multivalued variables

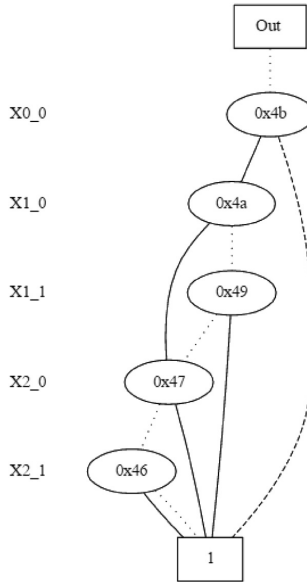


Figure 11.1 BDD for query *pandemic* in the *epidemic.pl* example, drawn using the CUDD function for exporting the BDD to the dot format of Graphviz.

Multivalued Variable Index	Rule Index	Grounding Substitution
0	1	[]
1	0	[david]
2	0	[robert]

and the clause groundings is encoded in the `Var` argument. For example, multivalued variable with index 1 is associated with the rule with index 0 (the first rule of the program) with grounding `_/david` and is encoded with two Boolean variables, `X1_0` and `X1_1`, since it can take three values. The hexadecimal numbers in the nodes are part of their memory address and are used to uniquely identify nodes.

11.4 Gaussian Processes

A Gaussian Process (GP) defines a probability distribution over functions [Bishop, 2016, Section 6.4]. This distribution has the property that, given N values, their image through a function sampled from the Gaussian process follows a multivariate normal with N dimensions, mean 0, and covariance

matrix K . In other words, if function $f(\mathbf{x})$ is sampled from a Gaussian process, then, for any finite selection of points $\mathbf{X} = \{\mathbf{x}^1, \dots, \mathbf{x}^N\}$, the density is

$$p(f(\mathbf{x}^1), \dots, f(\mathbf{x}^N)) = \mathcal{N}(\mathbf{0}, \mathbf{K}),$$

i.e., it is a Gaussian with mean 0 and covariance matrix \mathbf{K} . A GP is defined by a kernel function k that determines \mathbf{K} as $\mathbf{K}_{ij} = k(\mathbf{x}^i, \mathbf{x}^j)$.

GPs can be used for regression: the random functions predict the y value corresponding to a x value using the model

$$y = f(x) + \epsilon$$

where ϵ is a random noise variable with variance s^2 .

Given sets (column vectors) $\mathbf{X} = (x^1, \dots, x^N)^T$ and $\mathbf{Y} = (y^1, \dots, y^N)^T$ of observed values, the task is to predict the y value for a new point x . It can be proved [Bishop, 2016, Equations (6.66) and (6.67)] that y is Gaussian distributed with mean and variance

$$\mu = \mathbf{k}^T \mathbf{C}^{-1} \mathbf{Y} \quad (11.1)$$

$$\sigma^2 = k(x, x) - \mathbf{k}^T \mathbf{C}^{-1} \mathbf{k} \quad (11.2)$$

where \mathbf{k} is the column vector with elements $k(x^i, x)$ and \mathbf{C} has elements $\mathbf{C}_{ij} = k(x^i, x^j) + s^2 \delta_{ij}$, with s^2 user defined (the variance that is assumed for the random noise in the linear regression model) and δ_{ij} the Kronecker function ($\delta_{ij} = 1$ if $i = j$ and 0 otherwise). So $\mathbf{C} = \mathbf{K} + s^2 \mathbf{I}$ and $\mathbf{C}^{-1} = \mathbf{K}^{-1} - \frac{s^2 \mathbf{K}^{-1} \mathbf{K}^{-1}}{s^2 + \mathbf{K}^{-1} \mathbf{K} \mathbf{K}^{-1}}$.

A popular choice of kernel is the squared exponential

$$k(x, x') = \sigma^2 \exp \left[\frac{-(x - x')^2}{2l^2} \right]$$

with parameters σ and l . The user can define a prior distribution over the parameters instead of choosing particular values. In the case, the kernel itself is a random function and the predictions of regression will be random as well.

The program below (<http://cplint.eu/e/gpr.pl>) can sample kernels (and thus functions) and compute the expected value of the predictions for a squared exponential kernel (defined by predicate `sq_exp_p/3`) with parameter l uniformly distributed in 1, 2, 3 and σ uniformly distributed in $[-2, 2]$.

Goal `gp(X, Kernel, Y)`, given a list of values X and a kernel name, returns in Y the list of values $f(x)$ where x belongs to X and f is a function sampled from the Gaussian process.

316 *cplint Examples*

Goal `compute_cov(X, Kernel, Var, C)` returns in `C` the matrix `C` defined above with $\text{Var}=s^2$. It is called by `gp/3` with $\text{Var}=0$ in order to return `K`

```
gp(X, Kernel, Y) :-
    compute_cov(X, Kernel, 0, C),
    gp(C, Y).
gp(Cov, Y):gaussian(Y, Mean, Cov) :-
    length(Cov, N),
    list0(N, Mean).

compute_cov(X, Kernel, Var, C) :-
    length(X, N),
    cov(X, N, Kernel, Var, CT, CND),
    transpose(CND, CNDT),
    matrix_sum(CT, CNDT, C).

cov([], _, _, [], []).
cov([XH|XT], N, Ker, Var, [KH|KY], [KHND|KYND]) :-
    length(XT, LX),
    N1 is N-LX-1,
    list0(N1, KH0),
    cov_row(XT, XH, Ker, KH1),
    call(Ker, XH, XH, KXH0),
    KXH is KXH0+Var,
    append([KH0, [KXH], KH1], KH),
    append([KH0, [0], KH1], KHND),
    cov(XT, N, Ker, Var, KY, KYND).

cov_row([], _, [], []).
cov_row([H|T], XH, Ker, [KH|KT]) :-
    call(Ker, H, XH, KH),
    cov_row(T, XH, Ker, KT).

sq_exp_p(X, XP, K) :-
    sigma(Sigma),
    l(L),
    K is Sigma^2*exp(-(X-XP)^2/2/(L^2)).

l(L):uniform(L, [1,2,3]).

sigma(Sigma):uniform(Sigma, -2, 2).
```

Here `list0(N, L)` is true if `L` is a list with `N` elements all 0. This program exploits the possibility offered by `cplint` of defining multivariate Gaussian distributions.

`gp_predict(XP, Kernel, Var, XT, YT, YP)`, given the points described by the lists `XT` and `YT`, a kernel, and a list of points `XP`, predicts y values of points with x values in `XP` and returns them in `YP`. The predictions are the mean of y given by Equation (11.1), with `Var` being the s^2 parameter:

```
gp_predict(XP, Kernel, Var, XT, YT, YP) :-
    compute_cov(XT, Kernel, Var, C),
    matrix_inversion(C, C_1),
    transpose([YT], YST),
    matrix_multiply(C_1, YST, C_1T),
    gp_predict_single(XP, Kernel, XT, C_1T, YP).

gp_predict_single([], _, _, _, []).
gp_predict_single([XH|XT], Kernel, X, C_1T, [YH|YT]) :-
    compute_k(X, XH, Kernel, K),
    matrix_multiply([K], C_1T, [[YH]]),
    gp_predict_single(XT, Kernel, X, C_1T, YT).

compute_k([], _, _, []).
compute_k([XH|XT], X, Ker, [HK|TK]) :-
    call(Ker, XH, X, HK),
    compute_k(XT, X, Ker, TK).
```

Since the kernel here is random, the predictions of `gp_predict/6` will be random as well.

By calling the query

```
?- numlist(0, 10, X),
    mc_sample_arg_first(gp(X, sq_exp_p, Y), 5, Y, L).
```

we get five functions sampled from the Gaussian process with a squared exponential kernel at points $X = [0, \dots, 10]$. An example of output is shown in Figure 11.2.

The query

```
?- numlist(0, 10, X),
    XT=[2.5, 6.5, 8.5],
    YT=[1, -0.8, 0.6],
    mc_lw_sample_arg(gp_predict(X, sq_exp_p,
    0.3, XT, YT, Y), gp(XT, Kernel, YT), 5, Y, L).
```

draws five functions with a squared exponential kernel predicting points with X values in $[0, \dots, 10]$ given the three couples of points $XT = [2.5, 6.5, 8.5]$, $YT = [1, -0.8, 0.6]$. The graph of Figure 11.3 shows three of the functions together with the given points.

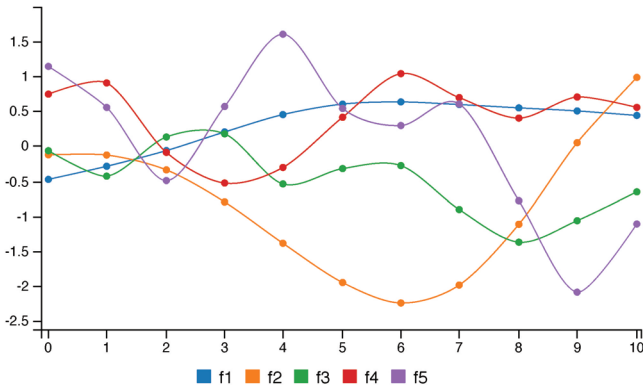


Figure 11.2 Functions sampled from a Gaussian process with a squared exponential kernel in `gpr.pl`.

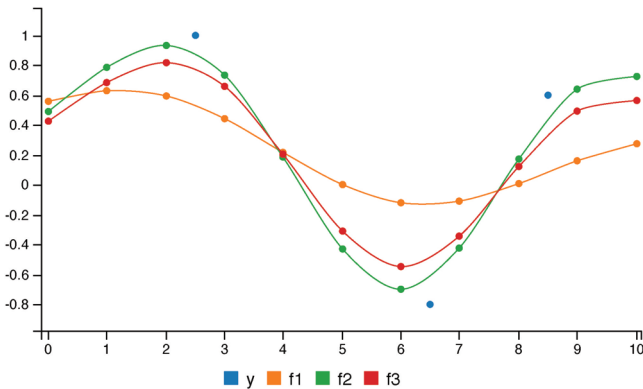


Figure 11.3 Functions from a Gaussian process predicting points with $X = [0, \dots, 10]$ with a squared exponential kernel in `gpr.pl`.

11.5 Dirichlet Processes

A Dirichlet Process (DP) [Teh, 2011] is a probability distribution whose range is itself a set of probability distributions. The DP is specified by a base distribution, which represents the expected value of the process. When sampling from a distribution in turn sampled from a DP, new samples have a non-zero probability of being equal to already sampled values. The process depends on a parameter α , called *concentration parameter*: with $\alpha \rightarrow 0$ a single value is sampled; with $\alpha \rightarrow \infty$, the distribution is equal to the base distribution. A DP with base distribution H and concentration parameter α is indicated with $DP(H, \alpha)$. A sample from $DP(H, \alpha)$ is a distribution P .

We are interested in sampling values from P . With abuse of terminology, we say that these values are sampled from the DP. There are several equivalent views of the DP, we present two of them in the following.

11.5.1 The Stick-Breaking Process

Example http://cplint.eu/e/dirichlet_process.pl encodes a view of DPs called *stick-breaking process*.

In this view, the procedure for sampling values from $DP(H, \alpha)$ can be described as follows. To sample the first value, a sample β_1 is taken from the beta distribution $Beta(1, \alpha)$ and a coin with heads probability equal to β_1 is flipped. If the coin lands on heads, a sample x_1 from the base distribution is taken and returned. Otherwise, a sample β_2 is taken again from $Beta(1, \alpha)$ and a coin is flipped. This procedure is repeated until heads are obtained, the index i of β_i being the index of the value x_i to be returned. The following values are sampled in a similar way, with the difference that, if for an index i , values x_i and β_i were already sampled, that value is returned.

This view is called stick-breaking because we can see the process as starting with a stick of length 1 which is progressively broken: first a piece β_1 long is broken off, then a piece β_2 long is broken off from the remaining piece, and so on. The length of the i -th piece is therefore

$$\prod_{k=1}^{i-1} (1 - \beta_k) \beta_i$$

and indicates the probability that the i -th sample x_i from the base distribution is returned. The smaller α is, the more probable high values of β_i are and the more often already sampled values are returned, yielding a more concentrated distribution.

In the example below, the base distribution is a Gaussian with mean 0 and variance 1, $\mathcal{N}(0, 1)$. The distribution of values is handled by predicates `dp_value(NV, Alpha, V)`, which returns (in V) the NV -th sample from the DP with concentration parameter `Alpha`, and `dp_n_values(N0, N, Alpha, L)`, which returns in L a list of $N-N0$ samples from the DP with concentration parameter `Alpha`.

The distribution of indexes is handled by predicate `dp_stick_index/4`.

```
dp_value(NV, Alpha, V) :-
    dp_stick_index(NV, Alpha, I),
    dp_pick_value(I, V).
```

320 *cplint* Examples

```
dp_pick_value (_, V) : gaussian(V, 0, 1) .

dp_stick_index (NV, Alpha, I) :-
  dp_stick_index (1, NV, Alpha, I) .
dp_stick_index (N, NV, Alpha, V) :-
  stick_proportion (N, Alpha, P) ,
  choose_prop (N, NV, Alpha, P, V) .

choose_prop (N, NV, _Alpha, P, N) :-
  pick_portion (N, NV, P) .
choose_prop (N, NV, Alpha, P, V) :-
  neg_pick_portion (N, NV, P) ,
  N1 is N+1,
  dp_stick_index (N1, NV, Alpha, V) .

stick_proportion (_, Alpha, P) : beta (P, 1, Alpha) .

pick_portion (_, _, P) : P ; neg_pick_portion (_, _, P) : 1-P .

dp_n_values (N, N, _Alpha, []) :- ! .

dp_n_values (N0, N, Alpha, [[V]-1|Vs]) :-
  N0 < N,
  dp_value (N0, Alpha, V) ,
  N1 is N0+1,
  dp_n_values (N1, N, Alpha, Vs) .
```

The query

```
?- mc_sample_arg (dp_stick_index (1, 10.0, V) , 2000, V, L) ,
  histogram (L, Chart, [nbins (100)]) .
```

draws the density of indexes with concentration parameter 10 using 2000 samples (see Figure 11.4).

The query

```
?- mc_sample_arg_first (dp_n_values (0, 2000, 10.0, V) , 1, V, L) ,
  L = [Vs-_] ,
  histogram (Vs, Chart, [nbins (100)]) .
```

draws the density of values over 2000 samples from a DP with concentration parameter 10 (see Figure 11.5).

The query

```
?- hist_repeated_indexes (1000, 100, G) .
```

called over the program:

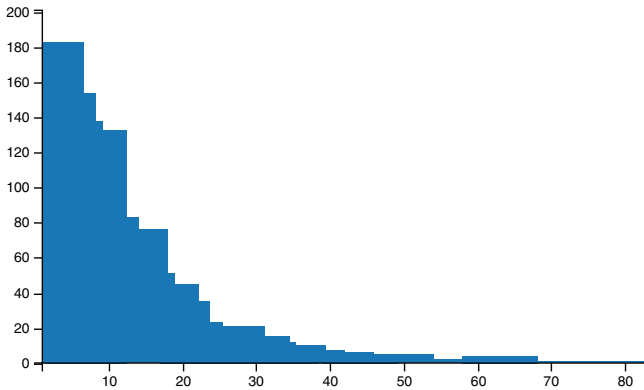


Figure 11.4 Distribution of indexes with concentration parameter 10 for the stick-breaking example `dirichlet_process.pl`.

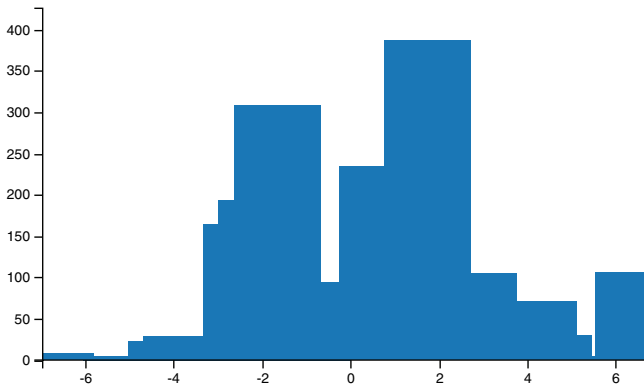


Figure 11.5 Distribution of values with concentration parameter 10 for the stick-breaking example `dirichlet_process.pl`.

```

hist_repeated_indexes (Samples, NBins, Chart) :-
    repeat_sample (0, Samples, L),
    histogram (L, Chart, [nbins (NBins)]).

repeat_sample (S, S, []) :- !.
repeat_sample (S0, S, [[N]-1|LS]) :-
    mc_sample_arg_first (dp_stick_index (1, 1, 10.0, V), 10, V, L),
    length (L, N),
    S1 is S0+1,
    repeat_sample (S1, S, LS).

```

shows the distribution of the number of unique indexes over 10 samples from a DP with concentration parameter 10 (see Figure 11.6).

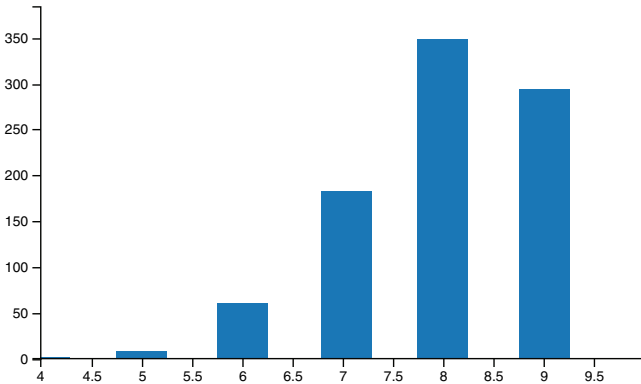


Figure 11.6 Distribution of unique indexes with concentration parameter 10 for the stick-breaking example `dirichlet_process.pl`.

11.5.2 The Chinese Restaurant Process

According to the Chinese restaurant view, a DP is a discrete-time stochastic process, analogous to seating customers at tables in a Chinese restaurant. When a new customer arrives at the restaurant, it is seated to a random table. It can be an existing table, chosen with a probability proportional to the number of clients already sitting at the table, or a new table, chosen with a probability proportional to α .

Formally, a sequence of samples x_1, x_2, \dots is drawn as follows. x_1 is drawn from the base distribution (corresponding to a new table as no customer is present). For $n > 1$, let $X^n = \{x^1, \dots, x^m\}$ be the set of distinct values previously sampled. x_n is set to a value $x^i \in X^n$ with probability $\frac{n_i}{\alpha + n - 1}$ where n_i is the number of previous observations $x_j, j < n$, such that $x_j = x^i$ (seating at an existing table), and is drawn from the base distribution with probability $\frac{\alpha}{\alpha + n - 1}$ (seating at a new table). Since

$$\sum_{i=1}^m \frac{n_i}{\alpha + n - 1} + \frac{\alpha}{\alpha + n - 1} = \frac{n - 1}{\alpha + n - 1} + \frac{\alpha}{\alpha + n - 1} = 1$$

this is a valid sampling process.

In example http://cplint.eu/e/dp_chinese.pl, the base distribution is a Gaussian with mean 0 and variance 1. Counts are kept and updated by predicate `update_counts/5`.

```
dp_n_values(N0, N, Alpha, [[V]-1|Vs], Counts0, Counts) :-
    N0 < N,
```

```

dp_value(N0,Alpha,Counts0,V,Counts1),
N1 is N0+1,
dp_n_values(N1,N,Alpha,Vs,Counts1,Counts).

dp_value(NV,Alpha,Counts,V,Counts1) :-
  draw_sample(Counts,NV,Alpha,I),
  update_counts(0,I,Alpha,Counts,Counts1),
  dp_pick_value(I,V).

update_counts(_I0,_I,Alpha,[_C],[1,Alpha]) :- !.
update_counts(I,I,_Alpha,[C|Rest],[C|Rest]) :-
  C1 is C+1.
update_counts(I0,I,Alpha,[C|Rest],[C|Rest1]) :-
  I1 is I0+1,
  update_counts(I1,I,Alpha,Rest,Rest1).

draw_sample(Counts,NV,Alpha,I) :-
  NS is NV+Alpha,
  maplist(div(NS),Counts,Probs),
  length(Counts,LC),
  numlist(1,LC,Values),
  maplist(pair,Values,Probs,Discrete),
  take_sample(NV,Discrete,I).

take_sample(_,D,V):discrete(V,D).

dp_pick_value(_,V):gaussian(V,0,1).

div(Den,V,P) :- P is V/Den.

pair(A,B,A:B).

```

Here `maplist/3` is a library predicate encoding the `maplist` primitive of functional programming: `maplist(Goal,List1,List2)` is true if `Goal` can be successfully applied to all couples of elements in the same position in the two lists.

The query

```

?- mc_sample_arg_first(dp_n_values(0,2000,10.0,V,[10.0],_),
  1,V,L),
  L=[Vs-],
  histogram(Vs,100,Chart).

```

draws the density of values over 2000 samples from a DP with concentration parameter 10. The resulting graph is similar to Figure 11.5.

11.5.3 Mixture Model

DPs can be used as a prior probability distribution in infinite mixture models. The objective is to build a mixture model without specifying in advance the number k of components. In example http://cplint.eu/e/dp_mix.pl, samples are drawn from a mixture of normal distributions whose parameters are defined by means of a DP. For each component, the variance is sampled from a gamma distribution and the mean is sampled from a Gaussian with mean 0 and variance 30 times the variance of the component. The program in this case is equivalent to the one encoding the stick-breaking example, except for the `dp_pick_value/3` predicate that is shown below:

```
dp_pick_value(I,NV,V) :-
    ivar(I,IV),
    Var is 1.0/IV,
    mean(I,Var,M),
    value(NV,M,Var,V).

ivar(_,IV):gamma(IV,1,0.1).

mean(_,V0,M):gaussian(M,0,V) :- V is V0*30.

value(_,M,V,Val):gaussian(Val,M,V).
```

Given a vector of observations `obs([-1,7,3])`, the queries

```
?- prior(1000,100,G).
?- post(1000,30,G).
```

called over the program

```
prior(Samples,NBins,Chart) :-
    mc_sample_arg_first(dp_n_values(0,Samples,10.0,V),1,V,L),
    L=[Vs-],
    histogram(Vs,Chart,[nbins(NBins)]).

post(Samples,NBins,Chart) :-
    obs(O),
    maplist(to_val,O,O1),
    length(O1,N),
    mc_lw_sample_arg_log(dp_value(0,10.0,T),
        dp_n_values(0,N,10.0,O1),Samples,T,L),
    maplist(keys,L,LW),
    min_list(LW,Min),
    maplist(exp(Min),L,L1),
    histogram(L1,Chart,[nbins(NBins),min(-8),max(15)]).
```



```
keys(_W, W) .
```

```
exp(Min, L-W, L-W1) :- W1 is exp(W-Min) .
```

```
to_val(V, [V]-1) .
```

draw the prior and the posterior densities, respectively, using 200 samples (Figures 11.7 and 11.8). Likelihood weighting is used because the evidence involves values for continuous random variables. `mc_lw_sample_arg_log/5` differs from `mc_lw_sample_arg/5` because it returns the natural logarithm of the weights, useful when the evidence is very unlikely.

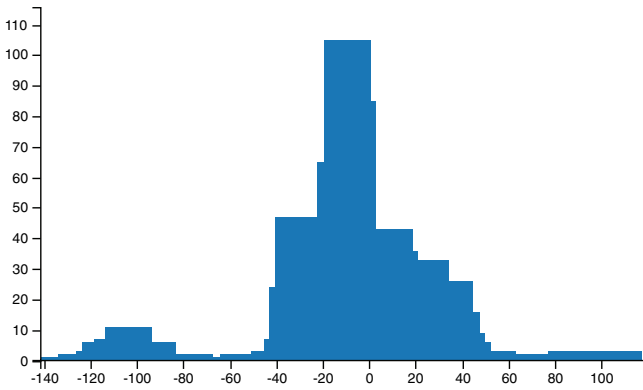


Figure 11.7 Prior density in the `dp_mix.pl` example.

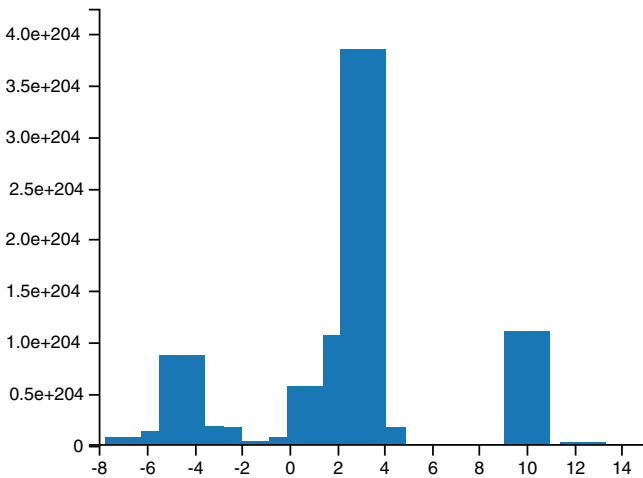


Figure 11.8 Posterior density in the `dp_mix.pl` example.

11.6 Bayesian Estimation

Let us consider a problem proposed for the Anglican system for probabilistic programming [Wood et al., 2014]³. We are trying to estimate the true value of a Gaussian-distributed random variable, given some observed data. The variance is known (its value is 2) and we suppose that the mean has itself a Gaussian distribution with mean 1 and variance 5. We take different measurements (e.g., at different times), indexed by an integer.

The program http://cplint.eu/e/gauss_mean_est.pl

```
val(I, X) :- mean(M), val(I, M, X).
mean(M) : gaussian(M, 1.0, 5.0).
val(_, M, X) : gaussian(X, M, 2.0).
```

models this problem.

Given that we observe 9 and 8 at indexes 1 and 2, how does the distribution of the random variable (value at index 0) change with respect to the case of no observations? This example shows that the parameters of the distribution atoms can be taken from the probabilistic atoms (`gaussian(X, M, 2.0)` and `value(_, M, X)` respectively). The query

```
?- mc_sample_arg(val(0, Y), 1000, Y, L0),
   mc_lw_sample_arg(val(0, X), (val(1, 9), val(2, 8)), 1000, X, L),
   densities(L0, L, Chart, [nbins(40)]).
```

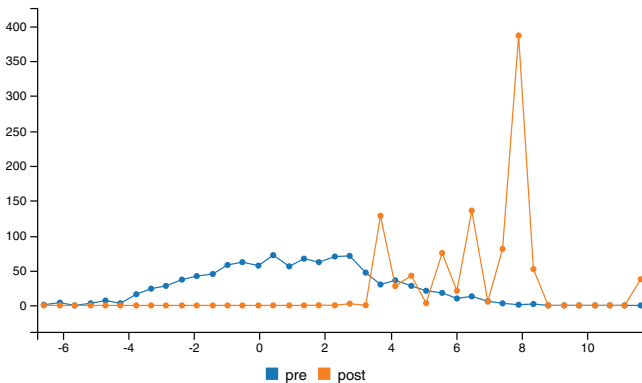


Figure 11.9 Prior and posterior densities in `gauss_mean_est.pl`.

³<https://bitbucket.org/probprog/anglican-examples/src/master/worksheets/gaussian-posteriors.clj>

takes 1000 samples of argument X of `val(0,X)` before and after the observation of `val(1,9)`, `val(2,8)` and draws the prior and posterior densities of the samples using a line chart. Figure 11.9 shows the resulting graph where the posterior is clearly peaked at around 8.

11.7 Kalman Filter

Example 59 represents a Kalman filter, i.e., a hidden Markov model with a real value as state and a real value as output. Program http://cplint.eu/e/kalman_filter.pl (adapted from [Nampally and Ramakrishnan, 2014]) encodes the example:

```
kf_fin(N,O,T) :-
    init(S),
    kf_part(0,N,S,O,T).

kf_part(I,N,S,[V|RO],T) :-
    I < N,
    NextI is I+1,
    trans(S,I,NextS),
    emit(NextS,I,V),
    kf_part(NextI,N,NextS,RO,T).
kf_part(N,N,S,[],S).

trans(S,I,NextS) :-
    {NextS := E+S},
    trans_err(I,E).

emit(NextS,I,V) :-
    {V := NextS+X},
    obs_err(I,X).

init(S):gaussian(S,0,1).

trans_err(_,E):gaussian(E,0,2).

obs_err(_,E):gaussian(E,0,1).
```

The next state is given by the current state plus Gaussian noise (with mean 0 and variance 2 in this example) and the output is given by the current state plus Gaussian noise (with mean 0 and variance 1 in this example). A Kalman filter can be considered as modeling a random walk of a single continuous state variable with noisy observations.

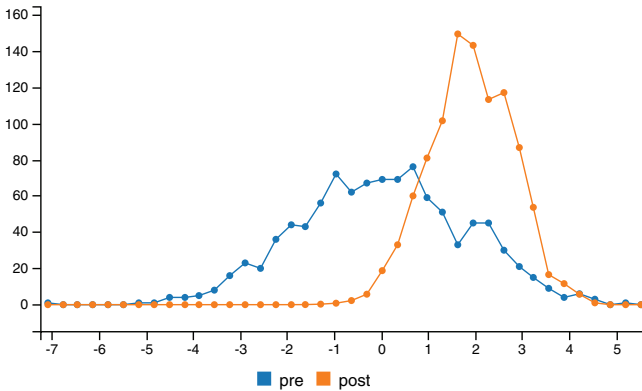


Figure 11.10 Prior and posterior densities in `kalman.pl`.

The goals $\{\text{NextS} ::= \text{E} + \text{S}\}$ and $\{\text{V} ::= \text{NextS} + \text{X}\}$ are CLP(R) constraints.

Given that, at time 0, the value 2.5 was observed, what is the distribution of the state at time 1 (filtering problem)? Likelihood weighting can be used to condition the distribution on evidence on a continuous random variable (evidence with probability 0). With CLP(R) constraints, it is possible to sample and to weight samples with the same program: when sampling, the constraint $\{\text{V} ::= \text{NextS} + \text{X}\}$ is used to compute V from X and NextS. When weighting, the constraint is used to compute X from V and NextS. The above query can be expressed with

```
?- mc_sample_arg(kf_fin(1,_O1,Y),1000,Y,L0),
   mc_lw_sample_arg(kf_fin(1,_O2,T),kf_fin(1,[2.5],_T),1000,
   T,L),densities(L0,L,Chart,[nbins(40)]).
```

that returns the graph of Figure 11.10, showing that the posterior distribution is peaked around 2.5.

Given a Kalman filter with four observations, the value of the state at those time points can be sampled by running particle filtering:

```
?- [O1,O2,O3,O4]=[-0.133, -1.183, -3.212, -4.586],
   mc_particle_sample_arg([kf_fin(1,T1),kf_fin(2,T2),
   kf_fin(3,T3),kf_fin(4,T4)], [kf_o(1,O1),kf_o(2,O2),
   kf_o(3,O3),kf_o(4,O4)],100,[T1,T2,T3,T4],
   [F1,F2,F3,F4]).
```

where `kf_o/2` is defined as

```
kf_o(N, ON) :-
    init(S),
    N1 is N-1,
    kf_part(0, N1, S, _O, _LS, T),
    emit(T, N, ON).
```

The list of samples is returned in `[F1, F2, F3, F4]`, with each element being the samples for a time point.

Given the true states from which the observations were obtained, Figure 11.11 shows a graph with the distributions of the state variable at time 1, 2, 3, and 4 (`S1, S2, S3, S4`, density on the left *Y*-axis) and with the points for the observations and the states with respect to time (time on the right *Y*-axis).

A two-dimensional Kalman filter can be used to track the movements of an object over a plane. For example,⁴ the object may perform a noisy circular motion. We receive noisy observations of the position and the objective is to estimate its position at the next time point. A Kalman filter may produce a 2-dimensional distribution of the next position of the object such as that shown in Figure 11.12, where the true and observed trajectories are shown in the upper part as red and green lines, respectively.

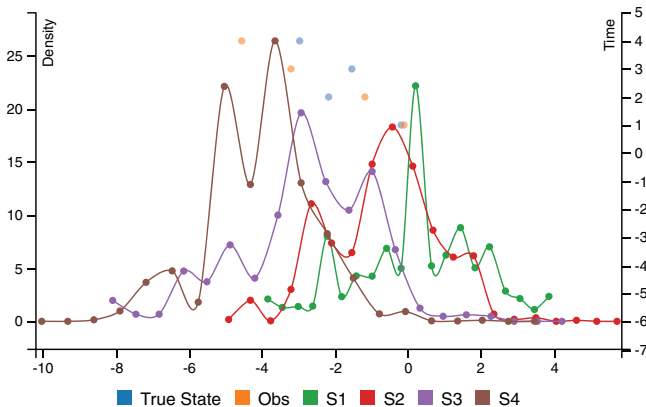


Figure 11.11 Example of particle filtering in `kalman.pl`.

⁴Inspired by <https://bitbucket.org/probprog/anglican-examples/src/master/worksheets/kalman.clj>.

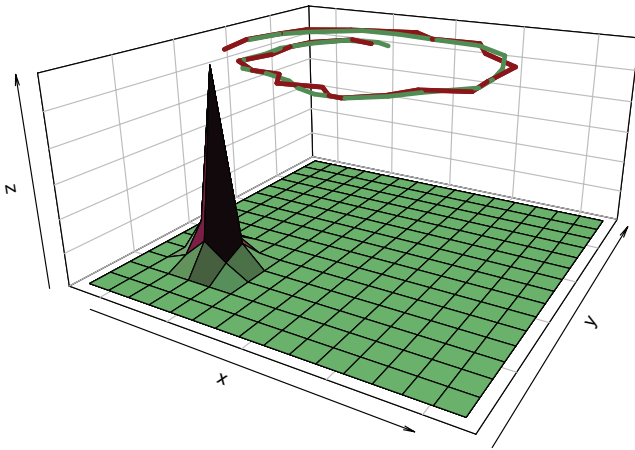


Figure 11.12 Particle filtering for a 2D Kalman filter.

11.8 Stochastic Logic Programs

SLPs (see Section 2.11.1) are used most commonly for defining a distribution over the values of arguments of a query. SLPs are a direct generalization of PCFGs and are particularly suitable for representing them. For example, the grammar

```
0.2:S->aS
0.2:S->bS
0.3:S->a
0.3:S->b
```

can be represented with the SLP

```
0.2::s([a|R]):-
  s(R).

0.2::s([b|R]):-
  s(R).

0.3::s([a]).

0.3::s([b]).
```

This SLP is encoded in *cplint* as program http://cplint.eu/e/slp_pcfg.pl:

```
s_as(N):0.2;s_bs(N):0.2;s_a(N):0.3;s_b(N):0.3.
```

```

s([a|R],N0):-
  s_as(N0),
  N1 is N0+1,
  s(R,N1).

s([b|R],N0):-
  s_bs(N0),
  N1 is N0+1,
  s(R,N1).

s([a],N0):-
  s_a(N0).

s([b],N0):-
  s_b(N0).

s(L):-
  s(L,0).

```

where the predicate $s/2$ has one more argument with respect to the SLP, which is used for passing a counter to ensure that different calls to $s/2$ are associated with independent random variables.

Inference with `cpaint` can then simulate the behavior of SLPs. For example, the query

```

?- mc_sample_arg_bar(s(S),100,S,P),
   argbar(P,C).

```

samples 100 sentences from the language and draws the bar chart of Figure 11.13.

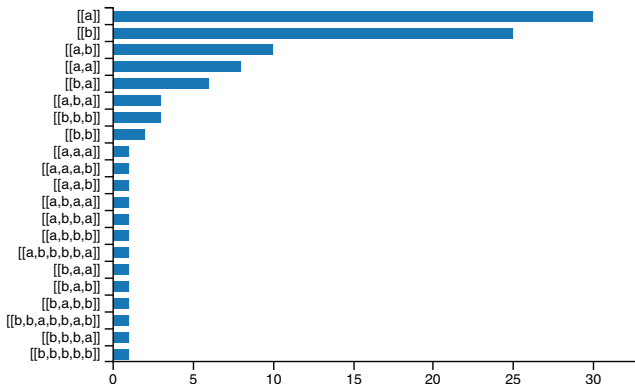


Figure 11.13 Samples of sentences of the language defined in `slp_pcfg.pl`.

11.9 Tile Map Generation

PLP can be used to generate random complex structures. For example, we can write programs for randomly generating maps of video games. Suppose that we are given a fixed set of tiles and we want to combine them to obtain a 2D map that is random but satisfies some soft constraints on the placement of tiles.

Suppose we want to draw a 10x10 map with a tendency to have a lake in the center. The tiles are randomly placed such that, in the central area, water is more probable. The problem can be modeled with the example http://cplint.eu/e/tile_map.swinb, where `map(H, W, M)` instantiates `M` to a map of height `H` and width `W`:

```
map(H,W,M):-
    tiles(Tiles),
    length(Rows,H),
    M=..[map,Tiles|Rows],
    foldl(select(H,W),Rows,1,_).

select(H,W,Row,N0,N):-
    length(RowL,W),
    N is N0+1,
    Row=..[row|RowL],
    foldl(pick_row(H,W,N0),RowL,1,_).

pick_row(H,W,N,T,M0,M):-
    M is M0+1,
    pick_tile(N,M0,H,W,T).
```

Here `foldl/4` is an SWI-Prolog [Wielemaker et al., 2012] library predicate that implements the `foldl` meta-primitive from functional programming: it aggregates the results of the application of a predicate to one or more lists. `foldl/4` is defined as:

```
foldl(P, [X11,...,X1n], [Xm1,...,Xmn], V0, Vn) :-
    P(X11, Xm1, V0, V1),
    ...
    P(X1n, Xmn, V', Vn).
```

`pick_tile(Y, X, H, W, T)` returns a tile for position (X, Y) of a map of size $W \times H$. The center tile is water:

```
pick_tile(HC,WC,H,W,water):-
    HC is H//2,
    WC is W//2,!.

```


In the central area, water is more probable:

```
pick_tile(Y,X,H,W,T):
    discrete(T, [grass:0.05,water:0.9,tree:0.025,rock:0.025]) :-
        central_area(Y,X,H,W), !
```

`central_area(Y,X,H,W)` is true if (X,Y) is adjacent to the center of the $W \times H$ map (definition omitted for brevity). In other places, tiles are chosen at random with distribution

```
[grass:0.5,water:0.3,tree:0.1,rock:0.1]:
```

```
pick_tile(_,_,_,_,T):
    discrete(T, [grass:0.5,water:0.3,tree:0.1,rock:0.1]).
```

We can generate a map by taking a sample of the query map $(10, 10, M)$ and collecting the value of M . For example, the map of Figure 11.14 can be obtained⁵.



Figure 11.14 A random tile map.

⁵Tiles from <https://github.com/silveira/openpixels>

11.10 Markov Logic Networks

We have seen in Section 2.12.2.1 that the MLN

```
1.5 Intelligent(x) => GoodMarks(x)
1.1 Friends(x, y) => (Intelligent(x) <=> Intelligent(y))
```

can be translated to the program below (<http://cplint.eu/e/inference/mln.swinb>):

```
clause1(X): 0.8175744762:- \+intelligent(X).
clause1(X): 0.1824255238:- intelligent(X), \+good_marks(X).
clause1(X): 0.8175744762:- intelligent(X), good_marks(X).

clause2(X,Y): 0.7502601056:-
  \+friends(X,Y).
clause2(X,Y): 0.7502601056:-
  friends(X,Y), intelligent(X),intelligent(Y).
clause2(X,Y): 0.7502601056:-
  friends(X,Y), \+intelligent(X),\+intelligent(Y).
clause2(X,Y): 0.2497398944:-
  friends(X,Y), intelligent(X),\+intelligent(Y).
clause2(X,Y): 0.2497398944:-
  friends(X,Y), \+intelligent(X),intelligent(Y).

intelligent(_):0.5.
good_marks(_):0.5.
friends(_,_):0.5.

student(anna).
student(bob).
```

The evidence must include the truth of all groundings of the `clausei` predicates:

```
evidence_mln:- clause1(anna), clause1(bob), clause2(anna,anna),
  clause2(anna,bob), clause2(bob,anna), clause2(bob,bob).
```

We have also evidence that Anna is friend with Bob and Bob is intelligent:

```
ev_intelligent_bob_friends_anna_bob :-
  intelligent(bob), friends(anna,bob),
  evidence_mln.
```

If we want to query the probability that Anna gets good marks given the evidence, we can ask:

```
?- prob(good_marks(anna),
  ev_intelligent_bob_friends_anna_bob,P).
```

while the prior probability of Anna getting good marks is given by:

?- prob(good_marks(anna), evidence_mln, P).

We obtain $P = 0.733$ from the first query and $P = 0.607$ from the second: given that Bob is intelligent and Anna is her friend, it is more probable that Anna gets good marks.

11.11 Truel

A truel [Kilgour and Brams, 1997] is a duel among three opponents. There are three truelists, a , b , and c , that take turns in shooting with a gun. The firing order is a , b , and c . Each truelist can shoot at another truelist or at the sky (deliberate miss). The truelists have these probabilities of hitting the target (if they are not aiming at the sky): $1/3$, $2/3$, and 1 for a , b , and c , respectively. The aim for each truelist is to kill all the other truelists. The question is: what should a do to maximize his probability of winning? Aim at b , c or the sky?

Let us see first the strategy for the other truelists and situations, following [Nguembang Fadja and Riguzzi, 2017]. When only two players are left, the best strategy is to shoot at the other player.

When all three players remain, the best strategy for b is to shoot at c , since if c shoots at him he is dead and if c shoots at a , b remains with c which is the best shooter. Similarly, when all three players remain, the best strategy for c is to shoot at b , since in this way, he remains with a , the worst shooter.

For a , it is more complex. Let us first compute the probability of a to win a duel with a single opponent. When a and c remain, a wins if it shoots c , with probability $1/3$. If he misses c , c will surely kill him. When a and b remain, the probability p of a winning can be computed with

$$\begin{aligned}
 p &= P(a \text{ hits } b) + P(a \text{ misses } b)P(b \text{ misses } b)p \\
 p &= \frac{1}{3} + \frac{2}{3} \times \frac{1}{3} \times p \\
 p &= \frac{3}{7}
 \end{aligned}$$

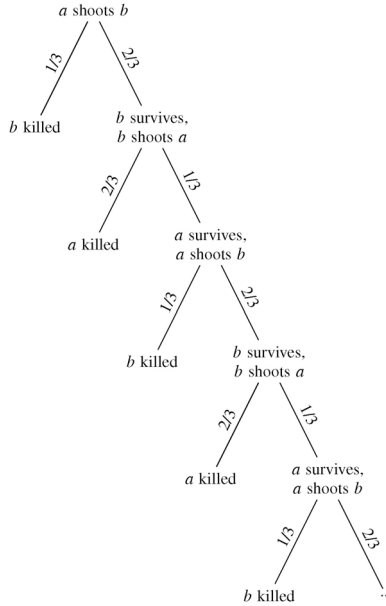


Figure 11.15 Probability tree of the truel with opponents *a* and *b*. From [Nguembang Fadja and Riguzzi, 2017].

The probability can also be computed by building the probability tree of Figure 11.15. The probability that *a* survives is thus

$$\begin{aligned}
 p &= \frac{1}{3} + \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} + \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} + \dots = \\
 &= \frac{1}{3} + \frac{2}{3^3} + \frac{2^2}{3^5} + \dots = \frac{1}{3} + \sum_{i=0}^{\infty} \frac{2}{3^3} \left(\frac{2}{9}\right)^i = \frac{1}{3} + \frac{\frac{2}{3^3}}{1 - \frac{2}{9}} = \\
 &= \frac{1}{3} + \frac{\frac{2}{3^3}}{\frac{7}{9}} = \frac{1}{3} + \frac{\frac{2}{3}}{7} = \frac{1}{3} + \frac{2}{21} = \frac{9}{21} = \frac{3}{7}
 \end{aligned}$$

When all three players remain, if *a* shoots at *b*, *b* is dead with probability 1/3 but then *c* will kill *a*. If *b* is not dead (probability 2/3), *b* shoots at *c* and kills him with probability 2/3. In this case, *a* is left in a duel with *b*, with a probability of surviving of 3/7. If *b* doesn't kill *c* (probability 1/3), *c* surely kills *b* and *a* is left in a duel with *c*, with a probability of surviving of 1/3. So overall, if *a* shoots at *b*, his probability of winning is

$$\frac{2}{3} \cdot \frac{2}{3} \cdot \frac{3}{7} + \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} = \frac{4}{21} + \frac{2}{27} = \frac{36 + 15}{189} = \frac{50}{189} \approx 0.2645$$

When all three players remain, if a shoots at c , c is dead with probability $1/3$. b then shoots at a and a survives with probability $1/3$ and a is then in a duel with b and surviving with probability $3/7$. If c survives (probability $2/3$), b shoots at c and kills him with probability $2/3$, so a remains in a duel with b and wins with probability $3/7$. If c survives again, he surely kills b and a is left in a duel with c , with a probability $1/3$ of winning. So overall, if a shoots at c , his probability of winning is

$$\frac{1}{3} \cdot \frac{1}{3} \cdot \frac{3}{7} + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{3}{7} + \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} = \frac{1}{21} + \frac{4}{21} + \frac{2}{27} = \frac{59}{189} \approx 0.3122$$

When all three players remain, if a shoots at the sky, b shoots at c and kills him with probability $2/3$, with a remaining in a duel with b . If b doesn't kill c , c surely kills b and a remains in a duel with c . So overall, if a shoots at the sky, his probability of winning is

$$\frac{2}{3} \cdot \frac{3}{7} + \frac{1}{3} \cdot \frac{1}{3} = \frac{2}{7} + \frac{1}{9} = \frac{25}{63} \approx 0.3968.$$

So the best strategy for a at the beginning of the game is to aim at the sky, contrary to intuition that would suggest trying to immediately eliminate one of the adversaries.

This problem can be modeled with an LPAD [Nguembang Fadja and Riguzzi, 2017]. However, as can be seen from Figure 11.15, the number of explanations may be infinite, so we need to use an appropriate exact inference algorithm, such as those discussed in Section 5.10, or a Monte Carlo inference algorithm. We discuss below the program <http://cplint.eu/e/truel.pl>. that uses MCINTYRE.

`survives_action(A, L0, T, S)` is true if A survives the truel performing action S with $L0$ still alive in turn T :

```
survives_action(A, L0, T, S) :-
    shoot(A, S, L0, T, L1),
    remaining(L1, A, Rest),
    survives_round(Rest, L1, A, T).
```

`shoot(H, S, L0, T, L)` is true when H shoots at S in round T with $L0$ and L the list of truelists still alive before and after the shot:

```
shoot(H, S, L0, T, L) :-
    (S=sky -> L=L0
    ;   (hit(T, H) -> delete(L0, S, L)
        ;   L=L0
        )
    ).
```

The probabilities of each truelist to hit the chosen target are

```
hit(_, a) : 1/3.
hit(_, b) : 2/3.
hit(_, c) : 1.
```

`survives(L, A, T)` is true if individual A survives the truel with truelists L at round T:

```
survives([A], A, _) :- !.
```

```
survives(L, A, T) :-
    survives_round(L, L, A, T).
```

`survives_round(Rest, L0, A, T)` is true if individual A survives the truel at round T with Rest still to shoot and L0 still alive:

```
survives_round([], L, A, T) :-
    survives(L, A, s(T)).
```

```
survives_round([H|_Rest], L0, A, T) :-
    base_best_strategy(H, L0, S),
    shoot(H, S, L0, T, L1),
    remaining(L1, H, Rest1),
    member(A, L1),
    survives_round(Rest1, L1, A, T).
```

The following strategies are easy to find:

```
base_best_strategy(b, [b, c], c).
base_best_strategy(c, [b, c], b).
base_best_strategy(a, [a, c], c).
base_best_strategy(c, [a, c], a).
base_best_strategy(a, [a, b], b).
base_best_strategy(b, [a, b], a).
base_best_strategy(b, [a, b, c], c).
base_best_strategy(c, [a, b, c], b).
```

Auxiliary predicate `remaining/3` is defined as

```
remaining([A|Rest], A, Rest) :- !.
remaining(_|Rest0], A, Rest) :-
    remaining(Rest0, A, Rest).
```

We can decide the best strategy for a by asking the probability of the queries

```
?- survives_action(a, [a, b, c], 0, b)
?- survives_action(a, [a, b, c], 0, c)
?- survives_action(a, [a, b, c], 0, sky)
```

By taking 1000 samples, we may get 0.256, 0.316, and 0.389, respectively, showing that *a* should aim at the sky.

11.12 Coupon Collector Problem

The coupon collector problem is described in [Kaminski et al., 2016] as

Suppose each box of cereal contains one of N different coupons and once a consumer has collected a coupon of each type, he can trade them for a prize. The aim of the problem is determining the average number of cereal boxes the consumer should buy to collect all coupon types, assuming that each coupon type occurs with the same probability in the cereal boxes.

If there are N different coupons, how many boxes, T , do I have to buy to get the prize? This problem is modeled by program <http://cplint.eu/e/coupon.swinb> defining predicate `coupons/2` such that goal `coupons(N, T)` is true if we need T boxes to get N coupons. The coupons are represented with a term for functor `cp/N` with the number of coupons as arity. The i -th argument of the term is 1 if the i -th coupon has been collected and is a variable otherwise. The term thus represents an array:

```
coupons(N, T) :-
    length(CP, N),
    CPTerm =.. [cp|CP],
    new_coupon(N, CPTerm, 0, N, T).
```

If 0 coupons remain to be collected, the collection ends:

```
new_coupon(0, _CP, T, _N, T).
```

If N_0 coupons remain to be collected, we collect one and recurse:

```
new_coupon(N0, CP, T0, N, T) :-
    N0 > 0,
    collect(CP, N, T0, T1),
    N1 is N0 - 1,
    new_coupon(N1, CP, T1, N, T).
```

`collect/4` collects one new coupon and updates the number of boxes bought:

```
collect(CP, N, T0, T) :-
    pick_a_box(T0, N, I),
    T1 is T0 + 1,
    arg(I, CP, CPI),
    (var(CPI) ->
        CPI = 1, T = T1
    );
    collect(CP, N, T1, T)
).
```

`pick_a_box/3` randomly picks a box and so a coupon type, an element from the list `[1..N]`:

```
pick_a_box(_,N,I):uniform(I,L):- numlist(1, N, L).
```

If there are five different coupons, we may ask:

- How many boxes do I have to buy to get the prize?
- What is the distribution of the number of boxes I have to buy to get the prize?
- What is the expected number of boxes I have to buy to get the prize?

To answer the first query, we can take a single sample for `coupons(5,T)`: in the sample, the query will succeed as `coupons/2` is a determinate predicate and the result will instantiate `T` to a specific value. For example, we may get `T=15`. Note that the maximum number of boxes to buy is unbounded but the case where we have to buy an infinite number of boxes has probability 0, so sampling will surely finish.

To compute the distribution on the number of boxes, we can take a number of samples, say 1000, and plot the number of times a value is obtained as a function of the value. By doing so, we may get the graph in Figure 11.16.

To compute the expected number of boxes, we can take a number of samples, say 100, of `coupons(5,T)`. Each sample will instantiate `T`. By summing all these values and dividing by 100, the number of samples, we can get an estimate of the expectation. For example, we may get a value of 11.47.

We can also plot the dependency of the expected number of boxes from the number of coupons, obtaining Figure 11.17. As observed in [Kaminski

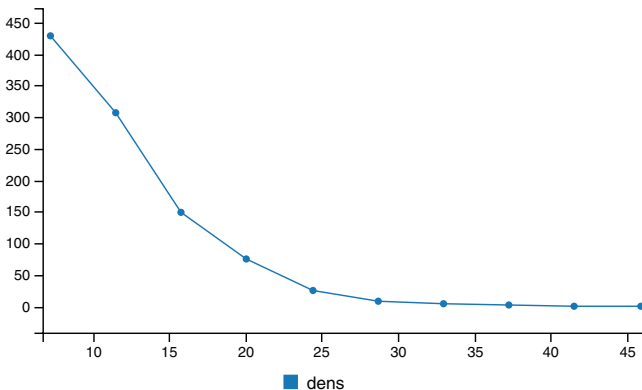


Figure 11.16 Distribution of the number of boxes.

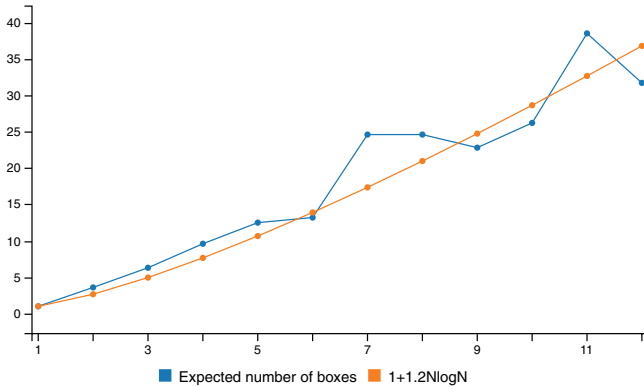


Figure 11.17 Expected number of boxes as a function of the number of coupons.

et al., 2016], the number of boxes grows as $O(N \log N)$ where N is the number of coupons. The graph also includes the curve $1 + 1.2N \log N$ that is similar to the first.

The coupon collector problem is similar to the sticker collector problem, where we have an album with a space for every different sticker, we can buy stickers in packs and our objective is to complete the album. A program for the coupon collector problem can be applied to solve the sticker collector problem: if you have N different stickers and packs contain P stickers, we can solve the coupon collector problem for N coupons and get the number of boxes T . Then the number of packs you have to buy to complete the collection is $\lceil T/P \rceil$. So we can write:

```
stickers(N,P,T):- coupons(N,T0), T is ceiling(T0/P).
```

If there are 50 different stickers and packs contain four stickers, by sampling the query `stickers(50, 4, T)`, we can get $T=47$, i.e., we have to buy 47 packs to complete the entire album.

11.13 One-Dimensional Random Walk

Let us consider the version of a random walk described in [Kaminski et al., 2016]: a particle starts at position $x = 10$ and moves with equal probability one unit to the left or one unit to the right in each turn. The random walk stops if the particle reaches position $x = 0$.

The walk terminates with probability 1 [Hurd, 2002] but requires, on average, an infinite time, i.e., the expected number of turns is infinite [Kaminski et al., 2016].

We can compute the number of turns with program http://cplint.eu/e/random_walk.swinb. The walk starts at time 0 and $x = 10$:

```
walk(T) :- walk(10, 0, T).
```

If x is 0, the walk ends; otherwise, the particle makes a move:

```
walk(0, T, T).
```

```
walk(X, T0, T) :-
    X > 0,
    move(T0, Move),
    T1 is T0 + 1,
    X1 is X + Move,
    walk(X1, T1, T).
```

The move is either one step to the left or to the right with equal probability.

```
move(T, 1) : 0.5; move(T, -1) : 0.5.
```

By sampling the query `walk(T)`, we obtain a success as `walk/1` is deterministic. The value for `T` represents the number of turns. For example, we may get `T = 3692`.

11.14 Latent Dirichlet Allocation

Text mining [Holzinger et al., 2014] aims at extracting knowledge from texts. Latent Dirichlet Allocation (LDA) [Blei et al., 2003] is a text mining technique which assigns topics to words in documents. Topics are taken from a finite set $\{1, \dots, K\}$. The model describes a generative process where documents are represented as random mixtures over latent topics and each topic defines a distribution over words. LDA assumes the following generative process for a corpus D consisting of M documents each of length N_i :

1. Sample θ_i from $\text{Dir}(\alpha)$, where $i \in \{1, \dots, M\}$ and $\text{Dir}(\alpha)$ is the Dirichlet distribution with parameter α .
2. Sample φ_k from $\text{Dir}(\beta)$, where $k \in \{1, \dots, K\}$.
3. For each of the word positions i, j , where $i \in \{1, \dots, M\}$ and $j \in \{1, \dots, N_i\}$:
 - (a) Sample a topic $z_{i,j}$ from $\text{Discrete}(\theta_i)$.
 - (b) Sample a word $w_{i,j}$ from $\text{Discrete}(\varphi_{z_{i,j}})$.

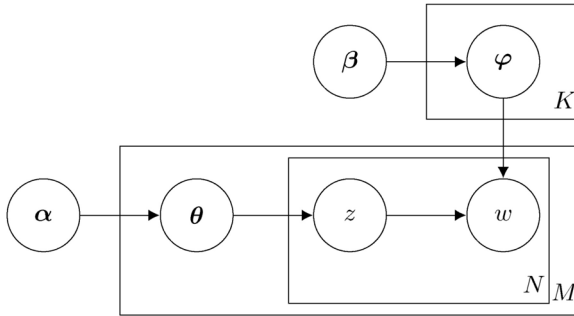


Figure 11.18 Smoothed LDA. From [Nguembang Fadja and Riguzzi, 2017].

This is a smoothed LDA model to be precise. Subscripts are often dropped, as in the plate diagrams in Figure 11.18.

The Dirichlet distribution is a continuous multivariate distribution whose parameter α is a vector $(\alpha_1, \dots, \alpha_K)$ and a value $\mathbf{x} = (x_1, \dots, x_K)$ sampled from $\text{Dir}(\alpha)$ is such that $x_j \in (0, 1)$ for $j = 1, \dots, K$ and $\sum_{j=1}^K x_j = 1$. A sample \mathbf{x} from a Dirichlet distribution can thus be the parameter for a discrete distribution $\text{Discrete}(\mathbf{x})$ with as many values as the components of \mathbf{x} : the distribution has $P(v_j) = x_j$ with v_j a value. Therefore, Dirichlet distributions are often used as priors for discrete distributions. The β vector above has V components where V is the number of distinct words.

The aim is to compute the probability distributions of words for each topic, of topics for each word, and the particular topic mixture of each document. This can be done with inference: the documents in the dataset represent the observations (evidence) and we want to compute the posterior distribution of the above quantities.

This problem can be modeled by the MCINTYRE program <http://cplint.eu/e/lda.swinb>, where predicate

```
word(Doc, Position, Word)
```

indicates that document `Doc` in position `Position` (from 1 to the number of words of the document) has word `Word` and predicate

```
topic(Doc, Position, Topic)
```

indicates that document `Doc` associates topic `Topic` to the word in position `Position`. We also assume that the distributions for both θ_i and φ_k are symmetric Dirichlet distributions with scalar concentration parameter η set using a fact for the predicate `eta/1`, i.e., $\alpha = [\eta, \dots, \eta]$ and $\beta = [\eta, \dots, \eta]$. The program is then:

344 *cplint* Examples

```
theta (_, Theta) : dirichlet (Theta, Alpha) :-
    alpha (Alpha) .

topic (DocumentID, _, Topic) : discrete (Topic, Dist) :-
    theta (DocumentID, Theta) ,
    topic_list (Topics) ,
    maplist (pair, Topics, Theta, Dist) .

word (DocumentID, WordID, Word) : discrete (Word, Dist) :-
    topic (DocumentID, WordID, Topic) ,
    beta (Topic, Beta) ,
    word_list (Words) ,
    maplist (pair, Words, Beta, Dist) .

beta (_, Beta) : dirichlet (Beta, Parameters) :-
    n_words (N) ,
    eta (Eta) ,
    findall (Eta, between (1, N, _) , Parameters) .

alpha (Alpha) :-
    eta (Eta) ,
    n_topics (N) ,
    findall (Eta, between (1, N, _) , Alpha) .

eta (2) .

pair (V, P, V:P) .
```

Suppose we have two topics, indicated with integers 1 and 2, and 10 words, indicated with integers 1, ..., 10:

```
topic_list (L) :-
    n_topics (N) ,
    numlist (1, N, L) .

word_list (L) :-
    n_words (N) ,
    numlist (1, N, L) .

n_topics (2) .

n_words (10) .
```

We can, for example, use the model generatively and sample values for the word in position 1 of document 1. The histogram of the frequency of word values when taking 100 samples is shown in Figure 11.19.

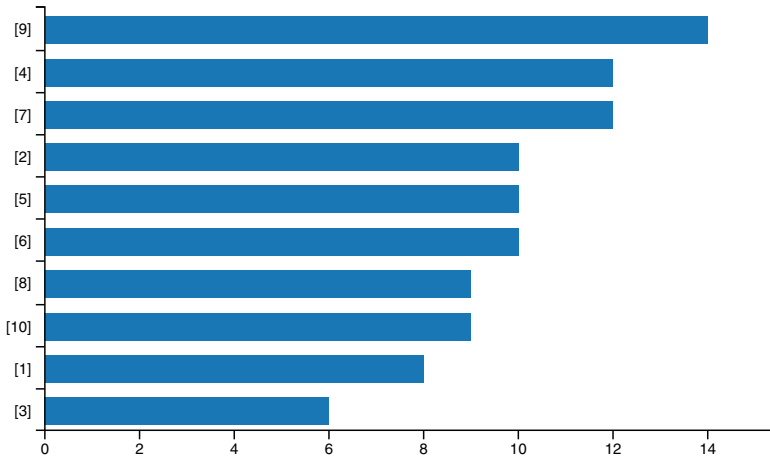


Figure 11.19 Values for word in position 1 of document 1.

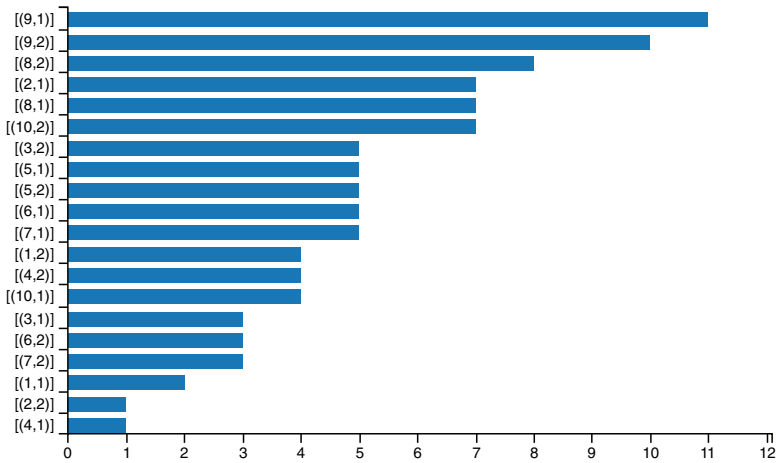


Figure 11.20 Values for couples (word,topic) in position 1 of document 1.

We can also sample values for couples (word, topic) in position 1 of document 1. The histogram of the frequency of the couples when taking 100 samples is shown in Figure 11.20.

We can use the model to classify the words into topics. Here we use conditional inference with Metropolis-Hastings. A priori both topics are about equally probable for word 1 of document, so if we take 100 samples of $\text{topic}(1, 1, T)$, we get the histogram in Figure 11.21. If we observe that words 1 and 2 of document 1 are equal ($\text{word}(1, 1, 1), \text{word}(1, 2, 1)$ as

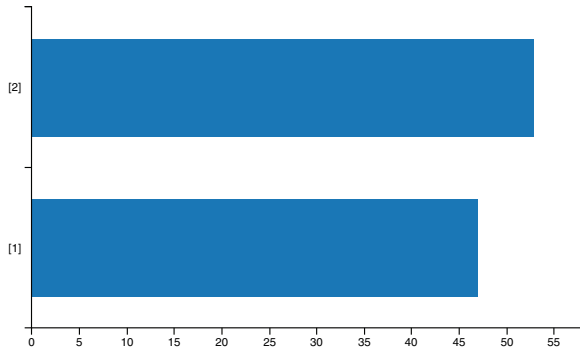


Figure 11.21 Prior distribution of topics for word in position 1 of document 1.

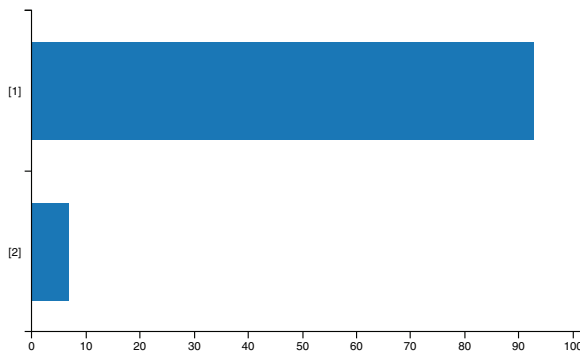


Figure 11.22 Posterior distribution of topics for word in position 1 of document 1.

evidence) and take again 100 samples, one of the topics gets more probable, as the histogram of Figure 11.22 shows. You can also see this if you look at the density of the probability of topic 1 before and after observing that words 1 and 2 of document 1 are equal: the observation makes the distribution less uniform, see Figure 11.23. `piercebayes` [Turliuc et al., 2016] is a PLP language that allows the specification of Dirichlet priors over discrete distribution. Writing LDA models with it is particularly simple.

11.15 The Indian GPA Problem

In the Indian GPA problem proposed by Stuart Russel [Perov et al., 2017; Nitti et al., 2016], the question is: if you observe that a student GPA is exactly 4.0, what is the probability that the student is from India, given that the American GPA score is from 0.0 to 4.0 and the Indian GPA score is from 0.0

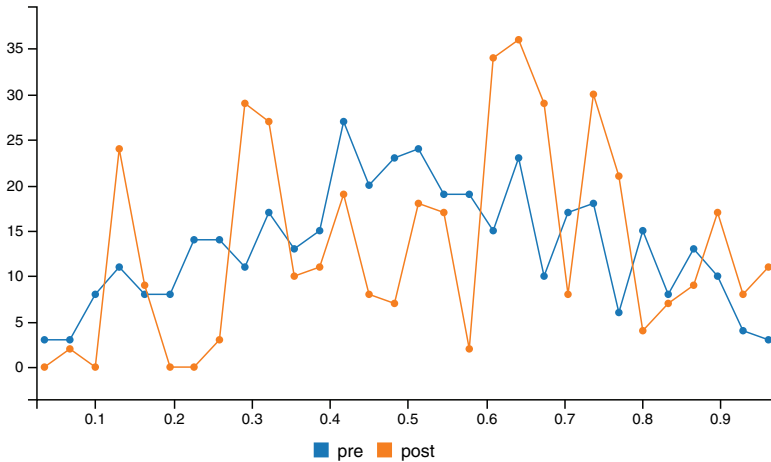


Figure 11.23 Density of the probability of topic 1 before and after observing that words 1 and 2 of document 1 are equal.

to 10.0? Stuart Russel observed that most probabilistic programming systems are not able to deal with this query because it requires combining continuous and discrete distributions. This problem can be modeled by building a mixture of a continuous and a discrete distribution for each nation to account for grade inflation (extreme values have a non-zero probability). Then the probability of the student's GPA is a mixture of the nation mixtures. Given this model and the fact that the student's GPA is exactly 4.0, the probability that the student is American is thus 1.0.

This problem can be modeled in MCINTYRE with program http://cplint.eu/e/indian_gpa.pl. The probability distribution of GPA scores for American students is continuous with probability 0.95 and discrete with probability 0.05:

```
is_density_A:0.95;is_discrete_A:0.05.
```

The GPA of an American student follows a beta distribution if the distribution is continuous:

```
agpa(A) : beta(A,8,2) :- is_density_A.
```

The GPA of an American student is 4.0 with probability 0.85 and 0.0 with probability 0.15 if the distribution is discrete:

```
american_gpa(G) : discrete(G,[4.0:0.85,0.0:0.15]) :-  
is_discrete_A.
```

or is obtained by rescaling the value of returned by `agpa/1` to the (0.0,4.0) interval:

```
american_gpa(A) :- agpa(A0), A is A0*4.0.
```

The probability distribution of GPA scores for Indian students is continuous with probability 0.99 and discrete with probability 0.01.

```
is_density_I : 0.99; is_discrete_I:0.01.
```

The GPA of an Indian student follows a beta distribution if the distribution is continuous:

```
igpa(I) : beta(I,5,5) :- is_density_I.
```

The GPA of an Indian student is 10.0 with probability 0.9 and 0.0 with probability 0.1 if the distribution is discrete:

```
indian_gpa(I) : discrete(I,[0.0:0.1,10.0:0.9]) :- is_discrete_I.
```

or is obtained by rescaling the value returned by `igpa/1` to the (0.0,10.0) interval:

```
indian_gpa(I) :- igpa(I0), I is I0*10.0.
```

The nation is America with probability 0.25 and India with probability 0.75.

```
nation(N) : discrete(N,[a:0.25,i:0.75]).
```

The GPA of the student is computed depending on the nation:

```
student_gpa(G) :- nation(a), american_gpa(G) .
```

```
student_gpa(G) :- nation(i), indian_gpa(G) .
```

If we query the probability that the nation is America given that the student got 4.0 in his GPA, we obtain 1.0, while the prior probability that the nation is America is 0.25.

11.16 Bongard Problems

The Bongard Problems Bongard [1970] were used in [De Raedt and Van Laer, 1995] as a testbed for ILP. Each problem consists of a number of pictures divided into two classes, positive and negative. The goal is to discriminate between the two classes.

The pictures contain geometric figures such as squares, triangles, and circles, with different properties, such as small, large, and pointing down, and different relationships between them, such as inside and above. Figure 11.24 shows some of these pictures.

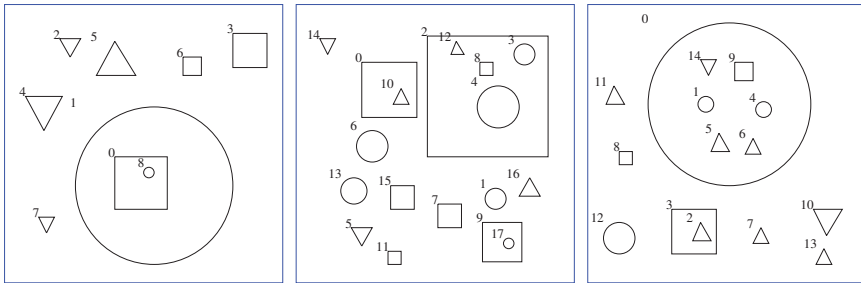


Figure 11.24 Bongard pictures.

A Bongard problem is encoded by http://cplint.eu/e/bongard_R.pl. Each picture is described by a mega-interpretation, in this case, contains a single example, either positive or negative. One such mega-interpretation can be

```
begin(model(2)).
pos.
triangle(o5).
config(o5,up).
square(o4).
in(o4,o5).
circle(o3).
triangle(o2).
config(o2,up).
in(o2,o3).
triangle(o1).
config(o1,up).
end(model(2)).
```

where `begin(model(2))` and `end(model(2))` denote the beginning and end of the mega-interpretation with identifier 2. The target predicate is `pos/0` that indicates the positive class. The mega-interpretation above includes one positive example.

Consider the input LPAD

```
pos:0.5 :-
    circle(A),
    in(B,A).
pos:0.5 :-
    circle(A),
    triangle(B).
```

and definitions for folds (sets of examples) such as

```
fold(train,[2,3,...]).
fold(test,[490,491,...]).
```

We can learn the parameters of the input program with EMBLEM using the query

```
induce_par([train],P).
```

The result is a program with updated values for the parameters:

```
pos:0.0841358 :-
  circle(A),
  in(B,A).
pos:0.412669 :-
  circle(A),
  triangle(B).
```

We can perform structure learning using SLIPCOVER by specifying a language bias:

```
modeh(*,pos).
modeb(*,triangle(-obj)).
modeb(*,square(-obj)).
modeb(*,circle(-obj)).
modeb(*,in(+obj,-obj)).
modeb(*,in(-obj,+obj)).
modeb(*,config(+obj,-#dir)).
```

Then the query

```
induce([train],P).
```

performs structure learning and returns a program:

```
pos:0.220015 :-
  triangle(A),
  config(A,down).
pos:0.12513 :-
  triangle(A),
  in(B,A).
pos:0.315854 :-
  triangle(A).
```

12

Conclusions

We have come to the end of our journey through probabilistic logic programming. I sincerely hope that I was able to communicate my enthusiasm for this field which combines the powerful results obtained in two previously separated fields: uncertainty in artificial intelligence and logic programming. PLP is growing fast but there is still much to do. An important open problem is how to scale the systems to large data, ideally of the size of Web, in order to exploit the data available on the Web, the Semantic Web, the so-called “knowledge graphs,” big databases such as Wikidata, and semantically annotated Web pages. Another important problem is how to deal with unstructured data such as natural language text, images, videos, and multimedia data in general.

For facing the scalability challenge, faster systems can be designed by exploiting symmetries in model using, for example, lifted inference, or restrictions can be imposed in order to obtain more tractable sublanguages. Another approach consists in exploiting modern computing infrastructures such as clusters and clouds and developing parallel algorithms, for example, using MapReduce [Riguzzi et al., 2016b].

For unstructured and multimedia data, handling continuous distributions effectively is fundamental. Inference for hybrid programs is relatively new but is already offered by various systems. The problem of learning hybrid programs, instead, is less explored, especially as regards structure learning. In domains with continuous random variables, neural networks and deep learning [Goodfellow et al., 2016] achieved impressive results. An interesting avenue for future work is how to exploit the techniques of deep learning for learning hybrid probabilistic logic programs.

Some works have already started to appear on the topic [Rocktäschel and Riedel, 2016; Yang et al., 2017; Nguembang Fadjia et al., 2017; Rocktäschel and Riedel, 2017; Evans and Grefenstette, 2018] but an encompassing framework dealing with different levels of certainty, complex relationships among entities, mixed discrete and continuous unstructured data, and extremely large size is still missing.

References

- M. Alberti, E. Bellodi, G. Cota, F. Riguzzi, and R. Zese. `cpInt` on SWISH: Probabilistic logical inference with a web browser. *Intelligenza Artificiale*, 11(1):47–64, 2017. doi: 10.3233/IA-170105.
- M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, and J. Zangari. The ASP system DLV2. In M. Balduccini and T. Janhunen, editors, *14th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2017)*, volume 10377 of *LNCS*. Springer, 2017. doi: 10.1007/978-3-319-61660-5_19.
- N. Angelopoulos. `clp(pdf(y))`: Constraints for probabilistic reasoning in logic programming. In F. Rossi, editor, *9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 784–788. Springer, 2003. doi: 10.1007/978-3-540-45193-8_53.
- N. Angelopoulos. Probabilistic space partitioning in constraint logic programming. In M. J. Maher, editor, *9th Asian Computing Science Conference (ASIAN 2004)*, volume 3321 of *LNCS*, pages 48–62. Springer, 2004. doi: 10.1007/978-3-540-30502-6_4.
- N. Angelopoulos. Notes on the implementation of FAM. In A. Hommersom and S. A. Abdallah, editors, *3rd International Workshop on Probabilistic Logic Programming (PLP 2016)*, volume 1661 of *CEUR Workshop Proceedings*, pages 46–58. CEUR-WS.org, 2016.
- K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991.
- K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19:9–71, 1994.
- R. Ash and C. Doléans-Dade. *Probability and Measure Theory*. Harcourt/Academic Press, 2000. ISBN 9780120652020.
- F. Bacchus. Using first-order probability logic for the construction of bayesian networks. In *9th Conference Conference on Uncertainty in Artificial Intelligence (UAI 1993)*, pages 219–226, 1993.

- R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997. doi: 10.1023/A:1008699807402.
- J. K. Baker. Trainable grammars for speech recognition. In D. H. Klatt and J. J. Wolf, editors, *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547–550, 1979.
- C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1):57–144, 2009. doi: 10.1017/S1471068408003645.
- L. Bauters, S. Schockaert, M. De Cock, and D. Vermeir. Possibilistic answer set programming revisited. In *26th International Conference on Uncertainty in Artificial Intelligence (UAI 2010)*. AUAI Press, 2010.
- V. Belle, G. V. den Broeck, and A. Passerini. Hashing-based approximate probabilistic inference in hybrid domains. In M. Meila and T. Heskes, editors, *31st International Conference on Uncertainty in Artificial Intelligence (UAI 2015)*, pages 141–150. AUAI Press, 2015a.
- V. Belle, A. Passerini, and G. V. den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In Q. Yang and M. Wooldridge, editors, *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 2770–2776. AAAI Press, 2015b.
- V. Belle, G. V. den Broeck, and A. Passerini. Component caching in hybrid domains with piecewise polynomial densities. In D. Schuurmans and M. P. Wellman, editors, *30th National Conference on Artificial Intelligence (AAAI 2015)*, pages 3369–3375. AAAI Press, 2016.
- E. Bellodi and F. Riguzzi. Experimentation of an expectation maximization algorithm for probabilistic logic programs. *Intelligenza Artificiale*, 8(1):3–18, 2012. doi: 10.3233/IA-2012-0027.
- E. Bellodi and F. Riguzzi. Expectation maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2):343–363, 2013.
- E. Bellodi and F. Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(2):169–212, 2015. doi: 10.1017/S1471068413000689.
- E. Bellodi, E. Lamma, F. Riguzzi, V. S. Costa, and R. Zese. Lifted variable elimination for probabilistic logic programming. *Theory and Practice of Logic Programming*, 14(4-5):681–695, 2014. doi: 10.1017/S1471068414000283.

- P. Berka. Guide to the financial data set. In *ECML/PKDD 2000 Discovery Challenge*, 2000.
- C. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2016. ISBN 9781493938438.
- D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- H. Blockeel. Probabilistic logical models for Mendel’s experiments: An exercise. In *Inductive Logic Programming (ILP 2004), Work in Progress Track*, pages 1–5, 2004.
- H. Blockeel and J. Struyf. Frankenstein classifiers: Some experiments on the Sisyphus data set. In *Workshop on Integration of Data Mining, Decision Support, and Meta-Learning (IDDM 2001)*, 2001.
- M. M. Bongard. *Pattern Recognition*. Hayden Book Co., Spartan Books, 1970.
- S. Bragaglia and F. Riguzzi. Approximate inference for logic programs with annotated disjunctions. In *21st International Conference on Inductive Logic Programming (ILP 2011)*, volume 6489 of *LNAI*, pages 30–37, Florence, Italy, 27–30 June 2011. Springer.
- D. Brannan. *A First Course in Mathematical Analysis*. Cambridge University Press, 2006. ISBN 9781139458955.
- R. Carnap. *Logical Foundations of Probability*. University of Chicago Press, 1950.
- M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.
- W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
- W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.
- Y. Chow and H. Teicher. *Probability Theory: Independence, Interchangeability, Martingales*. Springer Texts in Statistics. Springer, 2012.
- K. L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.
- P. Cohn. *Basic Algebra: Groups, Rings, and Fields*. Springer, 2003.
- A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un système de communication homme-machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d’Aix-Marseille, 1973.
- J. Côte-Real, T. Mantadelis, I. de Castro Dutra, R. Rocha, and E. S. Burnside. SKILL - A stochastic inductive logic learner. In T. Li, L. A. Kurgan,

- V. Palade, R. Goebel, A. Holzinger, K. Verspoor, and M. A. Wani, editors, *14th IEEE International Conference on Machine Learning and Applications (ICMLA 2015)*, pages 555–558. IEEE Press, 2015. doi: 10.1109/ICMLA.2015.159.
- J. Côrte-Real, I. de Castro Dutra, and R. Rocha. Estimation-based search space traversal in PILP environments. In J. Cussens and A. Russo, editors, *26th International Conference on Inductive Logic Programming (ILP 2016)*, volume 10326 of *LNCS*, pages 1–13. Springer, 2017. doi: 10.1007/978-3-319-63342-8_1.
- V. S. Costa, D. Page, M. Qazi, and J. Cussens. CLP(BN): Constraint logic programming for probabilistic knowledge. In *19th International Conference on Uncertainty in Artificial Intelligence (UAI 2003)*, pages 517–524. Morgan Kaufmann Publishers, 2003.
- F. G. Cozman and D. D. Mauá. On the semantics and complexity of probabilistic logic programs. *Journal of Artificial Intelligence Research*, 60:221–262, 2017.
- J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001. doi: 10.1023/A:1010924021315.
- E. Dantsin. Probabilistic logic programs and their semantics. In *Russian Conference on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.
- A. Darwiche. A logical approach to factoring belief networks. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. Williams, editors, *8th International Conference on Principles and Knowledge Representation and Reasoning*, pages 409–420. Morgan Kaufmann, 2002.
- A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In R. L. de Mántaras and L. Saitta, editors, *16th European Conference on Artificial Intelligence (ECAI 20014)*, pages 328–332. IOS Press, 2004.
- A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.
- A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In T. Walsh, editor, *22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 819–826. AAAI Press/IJCAI, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-143.
- A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
- J. Davis and M. Goadrich. The relationship between precision-recall and ROC curves. In *European Conference on Machine Learning (ECML 2006)*, pages 233–240. ACM, 2006.

- J. Davis, E. S. Burnside, I. de Castro Dutra, D. Page, and V. S. Costa. An integrated approach to learning bayesian networks of rules. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *European Conference on Machine Learning (ECML 2005)*, volume 3720 of *LNCS*, pages 84–95. Springer, 2005. doi: 10.1007/11564096_13.
- L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.
- L. De Raedt and I. Thon. Probabilistic rule learning. In P. Frasconi and F. A. Lisi, editors, *20th International Conference on Inductive Logic Programming (ILP 2010)*, volume 6489 of *LNCS*, pages 47–58. Springer, 2011. doi: 10.1007/978-3-642-21295-6_9.
- L. De Raedt and W. Van Laer. Inductive constraint logic. In *6th Conference on Algorithmic Learning Theory (ALT 1995)*, volume 997 of *LNAI*, pages 80–94. Springer, 1995.
- L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In M. M. Veloso, editor, *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, volume 7, pages 2462–2467. AAAI Press/IJCAI, 2007.
- L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS 2008 Workshop on Probabilistic Programming*, 2008.
- L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, 2008. Springer. ISBN 978-3-540-78651-1.
- L. De Raedt, K. Kersting, A. Kimmig, K. Revoredo, and H. Toivonen. Compressing probabilistic Prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.
- R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In L. P. Kaelbling and A. Saffiotti, editors, *19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1319–1325. Professional Book Center, 2005.
- A. Dekhtyar and V. Subrahmanian. Hybrid probabilistic programs. *Journal of Logic Programming*, 43(2):187–250, 2000.
- A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, 39(1):1–38, 1977.

- A. Dries, A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, and L. De Raedt. ProbLog2: Probabilistic logic programming. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2015)*, volume 9286 of *LNCS*, pages 312–315. Springer, 2015. doi: 10.1007/978-3-319-23461-8_37.
- D. Dubois and H. Prade. Possibilistic logic: a retrospective and prospective view. *Fuzzy Sets and Systems*, 144(1):3–23, 2004.
- D. Dubois, J. Lang, and H. Prade. Towards possibilistic logic programming. In *8th International Conference on Logic Programming (ICLP 1991)*, pages 581–595, 1991.
- D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming, vol. 3*, pages 439–514. Oxford University Press, 1994.
- S. Dzeroski. Handling imperfect data in inductive logic programming. In *4th Scandinavian Conference on Artificial Intelligence (SCAI 1993)*, pages 111–125, 1993.
- R. Evans and E. Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018. doi: 10.1613/jair.5714.
- F. Fages. Consistency of Clark’s completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.
- R. Fagin and J. Y. Halpern. Reasoning about knowledge and probability. *Journal of the ACM*, 41(2):340–367, 1994. doi: 10.1145/174652.174658.
- D. Fierens, G. Van den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.
- N. Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51:95–110, 2000.
- H. Gaifman. Concerning measures in first order calculi. *Israel Journal of Mathematics*, 2:1–18, 1964.
- M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011. doi: 10.3233/AIC-2011-0491.
- M. Gelfond and V. 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.
- G. Gerla. *Fuzzy Logic*, volume 11 of *Trends in Logic*. Springer, 2001. doi: 10.1007/978-94-015-9660-2_8.
- V. Gogate and P. M. Domingos. Probabilistic theorem proving. In F. G. Cozman and A. Pfeffer, editors, *27th International Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pages 256–265. AUAI Press, 2011.
- T. Gomes and V. S. Costa. Evaluating inference algorithms for the prolog factor language. In F. Riguzzi and F. Železný, editors, *21st International Conference on Inductive Logic Programming (ILP 2012)*, volume 7842 of *LNCS*, pages 74–85. Springer, 2012.
- I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT Press, 2016.
- N. D. Goodman and J. B. Tenenbaum. Inducing arithmetic functions, 2018. <http://forestdb.org/models/arithmetic.html>, accessed January 5, 2018.
- A. Gorlin, C. R. Ramakrishnan, and S. A. Smolka. Model checking with probabilistic tabled logic programming. *Theory and Practice of Logic Programming*, 12(4-5):681–700, 2012.
- P. Grünwald and J. Y. Halpern. Updating probabilities. *Journal of Artificial Intelligence Research*, 19:243–278, 2003. doi: 10.1613/jair.1164.
- B. Gutmann. *On continuous distributions and parameter estimation in probabilistic logic programs*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2011.
- B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter learning in probabilistic databases: A least squares approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008)*, volume 5211 of *LNCS*, pages 473–488. Springer, 2008.
- B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter estimation in ProbLog from annotated queries. Technical Report CW 583, KU Leuven, 2010.
- B. Gutmann, M. Jaeger, and L. De Raedt. Extending problog with continuous distributions. In P. Frasconi and F. A. Lisi, editors, *20th International Conference on Inductive Logic Programming (ILP 2010)*, volume 6489 of *LNCS*, pages 76–91. Springer, 2011a. doi: 10.1007/978-3-642-21295-6_12.
- B. Gutmann, I. Thon, and L. De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In D. Gunopulos,

- T. Hofmann, D. Malerba, and M. Vazirgiannis, editors, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2011)*, volume 6911 of *LNCS*, pages 581–596. Springer, 2011b.
- B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 11(4-5):663–680, 2011c.
- Z. Gyenis, G. Hofer-Szabo, and M. Rédei. Conditioning using conditional expectations: the Borel–Kolmogorov paradox. *Synthese*, 194(7): 2595–2630, 2017.
- S. Hadjichristodoulou and D. S. Warren. Probabilistic logic programming with well-founded negation. In D. M. Miller and V. C. Gaudet, editors, *42nd IEEE International Symposium on Multiple-Valued Logic, (ISMVL 2012)*, pages 232–237. IEEE Computer Society, 2012. doi: 10.1109/ISMVL.2012.26.
- J. Halpern. *Reasoning About Uncertainty*. MIT Press, 2003.
- J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990.
- A. C. Harvey. *Forecasting, structural time series models and the Kalman filter*. Cambridge University Press, 1990.
- J. Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, 1930.
- P. Hitzler and A. Seda. *Mathematical Aspects of Logic Programming Semantics*. Chapman & Hall/CRC Studies in Informatics Series. CRC Press, 2016.
- A. Holzinger, J. Schantl, M. Schroettner, C. Seifert, and K. Verspoor. Biomedical text mining: State-of-the-art, open problems and future challenges. In A. Holzinger and I. Jurisica, editors, *Interactive Knowledge Discovery and Data Mining in Biomedical Informatics*, volume 8401 of *LNCS*, pages 271–300. Springer, 2014. doi: 10.1007/978-3-662-43968-5_16.
- J. Hurd. A formal approach to probabilistic termination. In V. Carreño, C. A. Muñoz, and S. Tahar, editors, *15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *LNCS*, pages 230–245. Springer, 2002. doi: 10.1007/3-540-45685-6_16.
- K. Inoue, T. Sato, M. Ishihata, Y. Kameya, and H. 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.

- M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the EM algorithm by BDDs. In *Late Breaking Papers of the 18th International Conference on Inductive Logic Programming (ILP 2008)*, pages 44–49, 2008a.
- M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the EM algorithm by BDDs. Technical Report TR08-0004, Dep. of Computer Science, Tokyo Institute of Technology, 2008b.
- M. A. Islam. *Inference and learning in probabilistic logic programs with continuous random variables*. PhD thesis, State University of New York at Stony Brook, 2012.
- M. A. Islam, C. Ramakrishnan, and I. Ramakrishnan. Parameter learning in PRISM programs with continuous random variables. *CoRR*, abs/1203.4287, 2012a.
- M. A. Islam, C. Ramakrishnan, and I. Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12:505–523, 2012b. ISSN 1475-3081.
- M. Jaeger. Reasoning about infinite random structures with relational bayesian networks. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 570–581. Morgan Kaufmann, 1998.
- M. Jaeger and G. Van den Broeck. Liftability of probabilistic inference: Upper and lower bounds. In *2nd International Workshop on Statistical Relational AI (StarAI 2012)*, pages 1–8, 2012.
- J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998. doi: 10.1016/S0743-1066(98)10002-X.
- T. Janhunen. Representing normal programs with clauses. In R. L. de Mántaras and L. Saitta, editors, *16th European Conference on Artificial Intelligence (ECAI 20014)*, pages 358–362. IOS Press, 2004.
- B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In P. Thiemann, editor, *25th European Symposium on Programming, on Programming Languages and Systems (ESOP 2016)*, volume 9632 of LNCS, pages 364–389. Springer, 2016. doi: 10.1007/978-3-662-49498-1_15.
- K. Kersting and L. De Raedt. Towards combining inductive logic programming with Bayesian networks. In *11th International Conference on Inductive Logic Programming (ILP 2001)*, volume 2157 of LNCS, pages 118–131, 2001.

- K. Kersting and L. De Raedt. Basic principles of learning Bayesian logic programs. In *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, pages 189–221. Springer, 2008.
- H. Khosravi, O. Schulte, J. Hu, and T. Gao. Learning compact Markov logic networks with decision trees. *Machine Learning*, 89(3):257–277, 2012.
- D. M. Kilgour and S. J. Brams. The truel. *Mathematics Magazine*, 70(5): 315–326, 1997.
- A. Kimmig. *A Probabilistic Prolog and its Applications*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2010.
- A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. 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, 9–13 December 2008.
- A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011a.
- A. Kimmig, G. V. den Broeck, and L. D. Raedt. An algebraic Prolog for reasoning about possible worlds. In W. Burgard and D. Roth, editors, *25th AAAI Conference on Artificial Intelligence (AAAI 2011)*. AAAI Press, 2011b.
- J. Kisynski and D. Poole. Lifted aggregation in directed first-order probabilistic models. In C. Boutilier, editor, *21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1922–1929, 2009a.
- J. Kisynski and D. Poole. Constraint processing in lifted probabilistic inference. In J. Bilmes and A. Y. Ng, editors, *25th International Conference on Uncertainty in Artificial Intelligence (UAI 2009)*, pages 293–302. AUAI Press, 2009b.
- B. Knaster and A. Tarski. Un théorème sur les fonctions d’ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1928.
- K. Knopp. *Theory and Application of Infinite Series*. Dover Books on Mathematics. Dover Publications, 1951.
- S. Kok and P. Domingos. Learning the structure of Markov logic networks. In L. De Raedt and S. Wrobel, editors, *22nd International Conference on Machine learning*, pages 441–448. ACM Press, 2005.
- D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, Cambridge, MA, 2009.
- E. Koutsofios, S. North, et al. Drawing graphs with dot. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, 1991.

- R. A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.
- R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986. doi: 10.1007/BF03037383.
- T. Lager. Spaghetti and HMMeatballs, 2018. <https://web.archive.org/web/20150619013510/http://www.ling.gu.se/~lager/Spaghetti/spaghetti.html>, accessed June 14, 2018, snapshot at the Internet Archive from June 6, 2015 of <http://www.ling.gu.se/~lager/Spaghetti/spaghetti.html>, no more accessible.
- L. J. Layne and S. Qiu. Prediction for compound activity in large drug datasets using efficient machine learning approaches. In M. Khosrow-Pour, editor, *International Conference of the Information Resources Management Association*, pages 57–61. Idea Group Publishing, 2005. doi: 10.4018/978-1-59140-822-2.ch014.
- N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. doi: 10.1145/1149114.1149117.
- J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987. ISBN 3-540-18199-7.
- T. Mantadelis and G. Janssens. Dedicated tabling for a probabilistic setting. In M. V. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010)*, volume 7 of *LIPICs*, pages 124–133. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. doi: 10.4230/LIPICs.ICLP.2010.124.
- J. McDermott and R. S. Forsyth. Diagnosing a disorder in a classification benchmark. *Pattern Recognition Letters*, 73:41–43, 2016. doi: 10.1016/j.patrec.2016.01.004.
- S. Michels. *Hybrid Probabilistic Logics: Theoretical Aspects, Algorithms and Experiments*. PhD thesis, Radboud University Nijmegen, 2016.
- S. Michels, A. Hommersom, P. J. F. Lucas, M. Velikova, and P. W. M. Koopman. Inference for a new probabilistic constraint logic. In F. Rossi, editor, *23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 2540–2546. AAAI Press/IJCAI, 2013.
- S. Michels, A. Hommersom, P. J. F. Lucas, and M. Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence*, 228:1–44, 2015. doi: 10.1016/j.artint.2015.06.008.

- S. Michels, A. Hommersom, and P. J. F. Lucas. Approximate probabilistic inference with bounded error for hybrid probabilistic logic programming. In S. Kambhampati, editor, *25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 3616–3622. AAAI Press/IJCAI, 2016.
- B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling. Lifted probabilistic inference with counting formulas. In D. Fox and C. P. Gomes, editors, *23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1062–1068. AAAI Press, 2008.
- T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.
- P. Morettin, A. Passerini, and R. Sebastiani. Efficient weighted model integration via SMT-based predicate abstraction. In C. Sierra, editor, *26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pages 720–728. IJCAI, 2017. doi: 10.24963/ijcai.2017/100.
- S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
- S. Muggleton. Learning stochastic logic programs. *Electronic Transaction on Artificial Intelligence*, 4(B):141–153, 2000a.
- S. Muggleton. Learning stochastic logic programs. In L. Getoor and D. 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, 2000b.
- S. Muggleton. Learning structure and parameters of stochastic logic programs. In S. Matwin and C. Sammut, editors, *12th International Conference on Inductive Logic Programming (ILP 2002)*, volume 2583 of *LNCS*, pages 198–206. Springer, 2003. doi: 10.1007/3-540-36468-4_13.
- S. Muggleton, J. C. A. Santos, and A. Tamaddoni-Nezhad. Toplog: ILP using a logic program declarative bias. In M. G. de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 687–692. Springer, 2008. doi: 10.1007/978-3-540-89982-2_58.
- S. Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.
- C. J. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In L. Kosseim and D. Inkpen, editors, *25th Canadian Conference on Artificial Intelligence, Canadian AI 2012*, volume 7310 of *LNCS*, pages 356–361. Springer, 2012. doi: 10.1007/978-3-642-30353-1_36.

- K. P. Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, 2012.
- A. Nampally and C. Ramakrishnan. Adaptive MCMC-based inference in probabilistic logic programs. *arXiv preprint arXiv:1403.6036*, 2014.
- R. T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.
- A. Nguembang Fadja and F. Riguzzi. Probabilistic logic programming in action. In A. Holzinger, R. Goebel, M. Ferri, and V. Palade, editors, *Towards Integrative Machine Learning and Knowledge Extraction*, volume 10344 of *LNCS*. Springer, 2017. doi: 10.1007/978-3-319-69775-8_5.
- A. Nguembang Fadja, E. Lamma, and F. Riguzzi. Deep probabilistic logic programming. In C. Theil Have and R. Zese, editors, *4th International Workshop on Probabilistic Logic Programming (PLP 2017)*, volume 1916 of *CEUR-WS*, pages 3–14. Sun SITE Central Europe, 2017.
- P. Nicolas, L. Garcia, I. Stéphan, and C. Lefèvre. Possibilistic uncertainty handling for answer set programming. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):139–181, 2006.
- J. C. Nieves, M. Osorio, and U. Cortés. Semantics for possibilistic disjunctive programs. In *9th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2007)*, volume 4483 of *LNCS*, pages 315–320. Springer, 2007.
- N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.
- M. Nishino, A. Yamamoto, and M. Nagata. A sparse parameter learning method for probabilistic logic programs. In *Statistical Relational Artificial Intelligence, Papers from the 2014 AAAI Workshop*, volume WS-14-13 of *AAAI Workshops*. AAAI Press, 2014.
- D. Nitti, T. De Laet, and L. De Raedt. Probabilistic logic programming for hybrid relational domains. *Machine Learning*, 103(3):407–449, 2016. ISSN 1573-0565. doi: 10.1007/s10994-016-5558-8.
- J. Nivre. Logic programming tools for probabilistic part-of-speech tagging. Master thesis, School of Mathematics and Systems Engineering, Växjö University, October 2000.
- M. Osorio and J. C. Nieves. Possibilistic well-founded semantics. In *8th Mexican International International Conference on Artificial Intelligence (MICA I 2009)*, volume 5845 of *LNCS*, pages 15–26. Springer, 2009.
- A. Paes, K. Revoredo, G. Zaverucha, and V. S. Costa. Probabilistic first-order theory revision from examples. In S. Kramer and B. Pfahringer, editors, *15th International Conference on Inductive Logic Programming (ILP 2005)*, volume 3625 of *LNCS*, pages 295–311. Springer, 2005. doi: 10.1007/11536314_18.

- A. Paes, K. Revoredo, G. Zaverucha, and V. S. Costa. PFORTE: revising probabilistic FOL theories. In J. S. Sichman, H. Coelho, and S. O. Rezende, editors, *2nd International Joint Conference, 10th Ibero-American Conference on AI, 18th Brazilian AI Symposium, IBERAMIA-SBIA 2006*, volume 4140 of *LNCS*, pages 441–450. Springer, 2006. doi: 10.1007/11874850_48.
- L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- Y. Perov, B. Paige, and F. Wood. The Indian GPA problem, 2017. <https://bitbucket.org/probprog/anglican-examples/src/master/worksheets/indian-gpa.clj>, accessed June 1, 2018.
- A. Pfeffer. *Practical Probabilistic Programming*. Manning Publications, 2016. ISBN 9781617292330.
- G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
- D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993a.
- D. Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3):377–400, 1993b.
- D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94:7–56, 1997.
- D. Poole. Abducing through negation as failure: Stable models within the independent choice logic. *Journal of Logic Programming*, 44(1-3):5–35, 2000.
- D. Poole. First-order probabilistic inference. In G. Gottlob and T. Walsh, editors, *18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 985–991. Morgan Kaufmann Publishers, 2003.
- D. Poole. The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, pages 222–243. Springer, 2008.
- T. C. Przymusiński. Perfect model semantics. In R. A. Kowalski and K. A. Bowen, editors, *5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, pages 1081–1096. MIT Press, 1988.

- T. C. Przymusiński. 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.
- J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990. doi: 10.1007/BF00117105.
- L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
- L. D. Raedt, A. Dries, I. Thon, G. V. den Broeck, and M. Verbeke. Inducing probabilistic relational rules from probabilistic examples. In Q. Yang and M. Wooldridge, editors, *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 1835–1843. AAAI Press, 2015.
- I. Razgon. On OBDDs for CNFs of bounded treewidth. In C. Baral, G. D. Giacomo, and T. Eiter, editors, *14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014)*. AAAI Press, 2014.
- J. Renkens, G. Van den Broeck, and S. Nijssen. k-optimal: a novel approximate inference algorithm for ProbLog. *Machine Learning*, 89(3):215–231, 2012. doi: 10.1007/s10994-012-5304-9.
- J. Renkens, A. Kimmig, G. Van den Broeck, and L. De Raedt. Explanation-based approximate weighted model counting for probabilistic logics. In *28th National Conference on Artificial Intelligence, AAAI’14, Québec City, Québec, Canada*, pages 2490–2496. AAAI Press, 2014.
- K. Revoredo and G. Zaverucha. Revision of first-order Bayesian classifiers. In S. Matwin and C. Sammut, editors, *12th International Conference on Inductive Logic Programming (ILP 2002)*, volume 2583 of *LNCS*, pages 223–237. Springer, 2002. doi: 10.1007/3-540-36468-4_15.
- F. Riguzzi. Learning logic programs with annotated disjunctions. In A. Srinivasan and R. King, editors, *14th International Conference on Inductive Logic Programming (ILP 2004)*, volume 3194 of *LNCS*, pages 270–287. Springer, Sept. 2004. doi: 10.1007/978-3-540-30109-7_21.
- F. Riguzzi. A top down interpreter for LPAD and CP-logic. In *10th Congress of the Italian Association for Artificial Intelligence, (AI*IA 2007)*, volume 4733 of *LNAI*, pages 109–120. Springer, 2007a. doi: 10.1007/978-3-540-74782-6_11.
- F. Riguzzi. ALLPAD: Approximate learning of logic programs with annotated disjunctions. In S. Muggleton and R. Otero, editors, *16th International Conference on Inductive Logic Programming (ILP 2006)*, volume 4455 of *LNAI*, pages 43–45. Springer, 2007b. doi: 10.1007/978-3-540-73847-3_11.

- F. Riguzzi. Inference with logic programs with annotated disjunctions under the well founded semantics. In *24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 667–771. Springer, 2008a. doi: 10.1007/978-3-540-89982-2_54.
- F. Riguzzi. ALLPAD: Approximate learning of logic programs with annotated disjunctions. *Machine Learning*, 70(2-3):207–223, 2008b. doi: 10.1007/s10994-007-5032-8.
- F. Riguzzi. Extended semantics and inference for the independent choice logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009. doi: 10.1093/jigpal/jzp025.
- F. Riguzzi. SLGAD resolution for inference on logic programs with annotated disjunctions. *Fundamenta Informaticae*, 102(3-4):429–466, Oct. 2010. doi: 10.3233/FI-2010-392.
- F. Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013. doi: 10.3233/FI-2013-847.
- F. Riguzzi. Speeding up inference for probabilistic logic programs. *The Computer Journal*, 57(3):347–363, 2014. doi: 10.1093/comjnl/bxt096.
- F. Riguzzi. The distribution semantics for normal programs with function symbols. *International Journal of Approximate Reasoning*, 77:1–19, 2016. doi: 10.1016/j.ijar.2016.05.005.
- F. Riguzzi and N. Di Mauro. Applying the information bottleneck to statistical relational learning. *Machine Learning*, 86(1):89–114, 2012. doi: 10.1007/s10994-011-5247-6.
- F. Riguzzi and T. 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. doi: 10.4230/LIPICs.ICLP.2010.162.
- F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5):433–449, 2011. doi: 10.1017/S147106841100010X.
- F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2):279–302, 2013. doi: 10.1017/S1471068411000664.
- F. Riguzzi and T. Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM Transactions on Computational Logic*, 15(4):32:1–32:38, 2014. ISSN 1529-3785. doi: 10.1145/2629337.

- F. Riguzzi and T. Swift. Probabilistic logic programming under the distribution semantics. In M. Kifer and Y. A. Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*. Association for Computing Machinery and Morgan & Claypool, 2018.
- F. Riguzzi, E. Bellodi, and R. Zese. A history of probabilistic inductive logic programming. *Frontiers in Robotics and AI*, 1(6), 2014. ISSN 2296-9144. doi: 10.3389/frobt.2014.00006.
- F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, and G. Cota. Probabilistic logic programming on the web. *Software: Practice and Experience*, 46(10):1381–1396, 10 2016a. doi: 10.1002/spe.2386.
- F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. Scaling structure learning of probabilistic logic programs by MapReduce. In M. Fox and G. Kaminka, editors, *22nd European Conference on Artificial Intelligence (ECAI 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1602–1603. IOS Press, 2016b. doi: 10.3233/978-1-61499-672-9-1602.
- F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *International Journal of Approximate Reasoning*, 80:313–333, 1 2017a. doi: 10.1016/j.ijar.2016.10.002.
- F. Riguzzi, E. Lamma, M. Alberti, E. Bellodi, R. Zese, and G. Cota. Probabilistic logic programming for natural language processing. In F. Chesani, P. Mello, and M. Milano, editors, *Workshop on Deep Understanding and Reasoning, URANIA 2016*, volume 1802 of *CEUR Workshop Proceedings*, pages 30–37. Sun SITE Central Europe, 2017b.
- J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. doi: 10.1145/321250.321253.
- T. Rocktäschel and S. Riedel. Learning knowledge base inference with neural theorem provers. In J. Pujara, T. Rocktäschel, D. Chen, and S. Singh, editors, *5th Workshop on Automated Knowledge Base Construction, AKBC@NAACL-HLT 2016, San Diego, CA, USA, June 17, 2016*, pages 45–50. The Association for Computer Linguistics, 2016.
- T. Rocktäschel and S. Riedel. End-to-end differentiable proving. *CoRR*, abs/1705.11040, 2017.
- B. Russell. Mathematical logic as based on the theory of types. In J. van Heikenoort, editor, *From Frege to Gödel*, pages 150–182. Harvard Univ. Press, 1967.
- T. P. Ryan. *Modern Engineering Statistics*. John Wiley & Sons, 2007.
- V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

- T. Sato. A statistical learning method for logic programs with distribution semantics. In L. Sterling, editor, *12th International Conference on Logic Programming (ICLP 1995)*, pages 715–729. MIT Press, 1995.
- T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *15th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, volume 97, pages 1330–1339, 1997.
- T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15: 391–454, 2001.
- T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling by PRISM. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *LNCS*, pages 118–155. Springer, 2008. doi: 10.1007/978-3-540-78652-8_5.
- T. Sato and K. Kubota. Viterbi training in PRISM. *Theory and Practice of Logic Programming*, 15(02):147–168, 2015.
- T. Sato and P. Meyer. Tabling for infinite probability computation. In A. Dovier and V. S. 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.
- T. Sato and P. Meyer. Infinite probability computation by cyclic explanation graphs. *Theory and Practice of Logic Programming*, 14:909–937, 11 2014. ISSN 1475-3081. doi: 10.1017/S1471068413000562.
- T. Sato, Y. Kameya, and K. Kurihara. Variational Bayes via propositionalized probability computation in PRISM. *Annals of Mathematics and Artificial Intelligence*, 54(1-3):135–158, 2008.
- T. Sato, N.-F. Zhou, Y. Kameya, Y. Izumi, K. Kubota, and R. Kojima. PRISM User’s Manual (Version 2.3), 2017. <http://rjida.meijo-u.ac.jp/prism/download/prism23.pdf>, accessed June 8, 2018.
- O. Schulte and H. Khosravi. Learning graphical models for relational data via lattice search. *Machine Learning*, 88(3):331–368, 2012.
- O. Schulte and K. Routley. Aggregating predictions vs. aggregating features for relational classification. In *IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2014)*, pages 121–128. IEEE, 2014.
- R. Schwitter. Learning effect axioms via probabilistic logic programming. In R. Rocha, T. C. Son, C. Mears, and N. Saeedloei, editors, *Technical Communications of the 33rd International Conference*

- on *Logic Programming (ICLP 2017)*, volume 58 of *OASICS*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi: 10.4230/OASICS.ICLP.2017.8.
- P. Sevón, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCS*, pages 35–49. Springer, 2006.
- G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.
- D. S. Shterionov, J. Renkens, J. Vlasselaer, A. Kimmig, W. Meert, and G. Janssens. The most probable explanation for probabilistic logic programs with annotated disjunctions. In J. Davis and J. Ramon, editors, *24th International Conference on Inductive Logic Programming (ILP 2014)*, volume 9046 of *LNCS*, pages 139–153. Springer, 2015. doi: 10.1007/978-3-319-23708-4_10.
- P. Singla and P. Domingos. Discriminative training of Markov logic networks. In *20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 868–873. AAAI Press/The MIT Press, 2005.
- F. Somenzi. *CUDD: CU Decision Diagram Package Release 3.0.0*. University of Colorado, 2015. URL <http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf>.
- A. Srinivasan. The aleph manual, 2007. <http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html>, accessed April 3, 2018.
- A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.
- A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrac and S. Dzeroski, editors, *7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNCS*, pages 273–287. Springer Berlin Heidelberg, 1997.
- S. Srivastava. *A Course on Borel Sets*. Graduate Texts in Mathematics. Springer, 2013.
- L. Steen and J. Seebach. *Counterexamples in Topology*. Dover Books on Mathematics. Dover Publications, 2013.
- L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. Logic programming. MIT Press, 1994. ISBN 9780262193382.
- C. Stolle, A. Karwath, and L. De Raedt. *Cassic'cl*: An integrated ILP system. In A. Hoffmann, H. Motoda, and T. Scheffer, editors, *8th International Conference on Discovery Science (DS 2005)*, volume 3735 of *LNCS*, pages 354–362. Springer, 2005.

- T. Swift and D. S. Warren. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012. doi: 10.1017/S1471068411000500.
- T. Syrjänen and I. Niemelä. The Smodels system. In T. Eiter, W. Faber, and M. Truszczynski, editors, *6th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR 2001)*, volume 2173 of *LNCS*. Springer, 2001. doi: 10.1007/3-540-45402-0_38.
- N. Taghipour, D. Fierens, J. Davis, and H. Blockeel. Lifted variable elimination: Decoupling the operators from the constraint language. *Journal of Artificial Intelligence Research*, 47:393–439, 2013.
- A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.
- Y. W. Teh. Dirichlet process. In *Encyclopedia of machine learning*, pages 280–287. Springer, 2011.
- A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multivalued decision algorithms. In *8th International Symposium on Multiple-Valued Logic*, pages 171–178. IEEE Computer Society Press, 1978.
- I. Thon, N. Landwehr, and L. D. Raedt. A simple model for sequences of relational state descriptions. In *European conference on Machine Learning and Knowledge Discovery in Databases*, volume 5212 of *LNCS*, pages 506–521. Springer, 2008. ISBN 978-3-540-87480-5.
- C. Turliuc, L. Dickens, A. Russo, and K. Broda. Probabilistic abductive logic programming using Dirichlet priors. *International Journal of Approximate Reasoning*, 78:223–240, 2016. doi: 10.1016/j.ijar.2016.07.001.
- G. Van den Broeck. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24 (NIPS 2011)*, pages 1386–1394, 2011.
- G. Van den Broeck. *Lifted Inference and Learning in Statistical Relational Models*. PhD thesis, Ph. D. Dissertation, KU Leuven, 2013.
- G. Van den Broeck, I. Thon, M. van Otterlo, and L. De Raedt. DTProbLog: A decision-theoretic probabilistic Prolog. In M. Fox and D. Poole, editors, *24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 1217–1222. AAAI Press, 2010.
- G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In T. Walsh, editor, *22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 2178–2185. IJCAI/AAAI, 2011.

- G. Van den Broeck, W. Meert, and A. Darwiche. Skolemization for weighted first-order model counting. In C. Baral, G. D. Giacomo, and T. Eiter, editors, *14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014)*, pages 111–120. AAAI Press, 2014.
- A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
- J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, KU Leuven, 2003.
- J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In B. Demoen and V. Lifschitz, editors, *24th International Conference on Logic Programming (ICLP 2004)*, volume 3131 of *LNCS*, pages 431–445. Springer, 2004. doi: 10.1007/978-3-540-27775-0_30.
- J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.
- J. Vlasselaer, J. Renkens, G. Van den Broeck, and L. De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *1st International Workshop on Probabilistic Logic Programming (PLP 2014)*, pages 1–10, 2014.
- J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. In *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 1852–1858, 2015.
- J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt. Tp-compilation for inference in probabilistic logic programs. *International Journal of Approximate Reasoning*, 78:15–32, 2016. doi: 10.1016/j.ijar.2016.06.009.
- J. Von Neumann. Various techniques used in connection with random digits. *National Bureau of Standard (U.S.), Applied Mathematics Series*, 12: 36–38, 1951.
- W. Y. Wang, K. Mazaitis, N. Lao, and W. W. Cohen. Efficient inference and learning in a large knowledge base. *Machine Learning*, 100(1):101–126, Jul 2015. doi: 10.1007/s10994-015-5488-x.
- M. P. Wellman, J. S. Breese, and R. P. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(1):35–53, 1992.
- J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. doi: 10.1017/S1471068411000494.

- J. Wielemaker, T. Lager, and F. Riguzzi. SWISH: SWI-Prolog for sharing. In S. Ellmauthaler and C. Schulz, editors, *International Workshop on User-Oriented Logic Programming (IULP 2015)*, 2015.
- S. Willard. *General Topology*. Addison-Wesley series in mathematics. Dover Publications, 1970.
- F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *17th International conference on Artificial Intelligence and Statistics (AISTAT 2014)*, pages 1024–1032, 2014.
- F. Yang, Z. Yang, and W. W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, pages 2316–2325, 2017.
- N. L. Zhang and D. Poole. A simple approach to bayesian network computations. In *10th Canadian Conference on Artificial Intelligence, Canadian AI 1994*, pages 171–178, 1994.
- N. L. Zhang and D. L. Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.

Index

- σ -algebra
 - product 30
- θ -subsumption 284
- #P-complete 146, 151
- #SAT 254

- A**
- accuracy 291
- acyclic graph 34
- acyclicity 207
- acyclicity 20
 - LPAD 58
- aggregation parfactor 205
- Aleph 284, 300
- algebra 23, 92
- algebraic decision diagram 244
- algebraic fact 251
- ALLPAD 265
 - structure learning problem 287
- ancestor 35
- Anglican 326
- annotated disjunctive clause 42
- Annotated Probabilistic Logic Programming 88
- answer subsumption 161
- answer set, *see* model, stable 21
- answer set programming 22
- answer subsumption 162
- ant substitution 6
- anytime algorithm 221, 232, 233
- approximate inference 213
- aProbLog 250

- argument sampling 331
- arithmetic circuit 168
- arity 4
- atom 4, 46
- atomic choice 46, 49, 91
- atomic formulas 4
- atomic normal program 210
- attach operation 310

- B**
- background predicate 296
- backward probability 274
- base distribution 318
- Baum-Welch algorithm 262
- Bayes' theorem 31
- Bayesian estimation 326
- Bayesian Logic Programs 76
- Bayesian network 34, 41, 55
- beam search 285, 291, 296, 300
- belief propagation 40
- beta distribution 127, 319
- binary decision diagram 152, 179, 181, 249, 254, 256, 269, 271, 305, 313
- binomial distribution 127, 216
- BIOMINE network 55
- bipartite graph 40
- bivariate Gaussian density 29
- Borel-Kolmogorov paradox 32
- bottom clause 285, 296
- bottom element 2, 16, 103
- bottom-up parsing 310

bottom-up reasoning 11
 branch-and-bound 216, 229

C

C-FOVE 201
 c2d 168
 Cantor's diagonal argument 92
 Cartesian product 126
 infinite 110
 consistent 112
 causal independence 200
 causal mechanism 42
 causal reasoning 42
 causality 66
 central limit theorem 216
 chain 36
 chain rule 34
 chi squared 295
 child random variable 34
 Chinese restaurant process 322
 Clark's completion 13, 165, 175, 207
 classification 33
 clause 5
 definite 5
 normal 6
 clause body 5
 clause head 5
 clique 38, 40
 closed-world assumption 275
 closure under countable union 100
 CLP(BN) 76
 CLP(R) 128
 collider 36
 compatibility condition,
 see consistency condition 111
 complete binary decision diagram
 272
 complete interpretation 251
 complete lattice 2, 16, 103

completeness 8, 107
 composite choice 46, 93
 concentration parameter 318
 conditional density 31
 conditional inference 224
 conditional probability 30, 46
 conditional probability table 34
 deterministic 35
 conditionally independent random
 variables 32, 38, 64
 confidence interval 216
 binomial proportion 216
 consistency condition 111
 consistent set of atomic choices 46
 constraint
 anti-monotonic 288
 monotonic 288
 context free grammar 309
 convergent random variable 199
 convergent series 186
 Cora 181
 countable set 23, 91, 113
 counting formulas 202
 coupon collector problem 339
 covariance matrix 29, 315
 covering loop 291, 300
 covering set of composite choices 96
 covering set of explanations 96, 146,
 178
 CP-logic 66
 semantics 67
 valid theory 70
 CPL1 239
 cplint 222, 305
 continuous variables 126
 credal semantics 72, 73
 credal sets 130
 CUDD 158, 245
 cumulative distribution 26

cut 12
 cyclic explanation graph 187
 cylinder sets 110

D

d-separation 35, 40
 Datalog 7
 decision facts 241
 decision problem 241
 decision variable 241
 decreasing sequence 3
 deep learning 351
 default negation 6
 deleted path 274
 deletion rule 274
 depth
 program 17
 depth bound 215
 deputy variable 200
 derivation 8, 275
 derivation variable 189
 descendant 35
 determinant of a matrix 29
 determinate predicate 235
 determinate program 236
 deterministic decomposable negation
 normal form 167
 decision 171
 ordering 172
 smooth 167
 strongly deterministic 174
 structured decomposability 174
 deterministic dependency 35
 diagnosis 33
 Dirichlet distribution 126, 343
 Dirichlet process 318, 324
 discrete topology 110
 discrete-time stochastic process 322
 disjoint statements 43

disjoint sum 255
 disjoint-sum problem 151
 distribution semantics 41, 45, 64, 92
 Distributional Clauses 118, 126, 129
 valid program 120
 DLV 22
 DPLL 207
 Dsharp 168
 DTProbLog 241
 duel 335
 dynamic clauses 221
 dynamic models 178
 dynamic programming 149, 154,
 157, 262, 268
 dynamically stratification 18

E

eigenvalues 29
 elimination order 200
 EMBLEM 270, 296, 300
 learning problem 270
 equality theory 14
 equivalent networks 39
 equivalent sets of composite choices
 93
 error bound 215
 event, *see* measurable set
 Event Calculus 304
 evidence 33, 334
 exclusive-or assumption 146, 178,
 194, 236, 306
 existential quantifier 208
 expectation 27, 121, 235, 341
 conditional 31
 expectation maximization 259, 270,
 281, 282
 expected cost 241
 expected reward 241
 expected sufficient statistics 282

explanation 96, 146, 231, 300
 infinite 221
 explanation sum 256
 expressive power 38, 41, 56
 extended PRISM 124, 129, 282
 extension of resolution 240

F

fact 5
 factor 36
 heterogeneous 200
 factor graph 40
 factorization, *see* factorized model
 factorized model 36
 family of nodes 37
 feature 37
 Figaro 88
 finite support condition 120
 finitely additive probability measure
 24
 finitely additive probability space 24,
 95
 finitely well-defined program 96
 first-order logic 4
 fixpoint 3, 234
 greatest 3, 17
 least 3, 12, 17, 105, 177
 flexible probabilities 64, 91
 FOIL 284, 291
 fork 36
 forward probability 274
 forward-backward 262
 FOVE 201
 fully-connected 38
 function symbol 186
 function symbols 41, 46, 91, 221

G

gamma distribution 124, 126

Gaussian density, *see* Gaussian
 distribution 27
 Gaussian distribution 124, 126, 189,
 224, 319, 322, 324, 326
 Gaussian mixture model 115, 119,
 125, 127, 238
 Gaussian noise 327
 Gaussian process 314
 GC-FOVE 201
 generality relation 284
 generative definitions 130
 generative exclusiveness
 condition 187
 geometric distribution 127
 global scoring function 291
 gradient 282
 gradient computation 255
 gradient descent 267
 grammar 309
 graphical model 34, 41, 146
 greatest lower bound 16, 103
 greedy search 296
 grounding 6, 46

H

Herbrand base 6
 Herbrand universe 6
 heuristic 291
 hidden Markov model 81, 147, 262,
 309
 hybrid ProbLog 115
 parameter learning 282
 hybrid program 223
 hypothetical derivation sequence 69

I

idempotent 256
 immediate consequence operator 11

- imprecise probability distributions
 - 130
 - increasing sequence 3
 - increasing sequence of sets 99
 - independent and identically distributed
 - 44
 - Independent Choice Logic 43, 56
 - independent random variables 32
 - independent-and assumption 146, 178, 194, 261
 - independent-or assumption 179
 - Indian GPA problem 346
 - indicator function 121
 - inductive definition 9, 10
 - inductive logic programming 288
 - inference 33
 - approximate 146
 - COND 145
 - CONDATOMS 163
 - DISTR 145, 234
 - EVID 145, 163
 - exact 145
 - EXP 234, 306
 - EXPEC 340
 - MAP 145
 - MPE 145, 150, 163, 186, 254
 - PROB 255
 - infinite mixture model 324
 - infinite product 91
 - infinite-dimensional measurable space
 - 112
 - infinitely often 98
 - inhibitor 198
 - input variable 285
 - inside probability 262
 - Inside-Outside algorithm 262
 - interpretation
 - Herbrand 7
 - three-valued 15
 - consistent 16
 - two-valued 7
 - iterated fixpoint 17, 104
 - iterative deepening 213, 233
- J**
- joint cumulative distribution 28
 - joint event 33
 - joint probability 28
 - joint probability density 28
 - joint probability distribution 28, 34, 39
- K**
- k-best 215, 249
 - k-optimal 229
 - Kalman filter 128, 327
 - kernel
 - function 315
 - squared exponential 315
 - Knaster–Tarski theorem 3
 - knowledge base model construction
 - 41, 76
 - knowledge compilation 146, 150, 151
 - knowledge graph 351
 - Kolmogorov consistency theorem 111
- L**
- labeling function 252
 - language bias 285, 296
 - latent Dirichlet allocation 342
 - learning from entailment 283
 - least unsure path in a graph 240
 - least upper bound 16, 103
 - Lebesgue measure 126
 - Lebesgue-measurable 120
 - left corner grammar 309

- LeProbLog 255, 267, 282
 - parameter learning problem 267
 - level mapping 20
 - LFI-ProbLog 280
 - lifted absorption 201
 - lifted inference 146, 195, 351
 - lifted multiplication 201
 - lifted sum-out 201
 - likelihood ratio statistics 295
 - likelihood weighting 224, 234, 307
 - limit of a sequence of sets 98
 - literal 5
 - negative 6, 13
 - positive 6
 - LLPAD 265, 287
 - parameter learning problem 266
 - local scoring function 291
 - local search 249
 - log likelihood 261, 296
 - log-linear model 38
 - logic program
 - definite 5
 - normal 6, 13
 - Logic Program with Annotated Disjunctions 42, 51, 66
 - logic programming 8
 - logical consequence, *see* logical entailment
 - logical entailment 7, 14
 - loop
 - negative 15
 - positive 15
 - lower bound 213, 233
 - lower bound of a set 1
 - greatest 1
 - LP² 202
- M**
- m-estimate 291, 295
 - mapping 3
 - MapReduce 351
 - marginalization 31, 41, 46
 - Markov blanket 36, 38
 - Markov chain 222
 - Markov chain Monte Carlo 222
 - Markov logic network 84, 334
 - Markov network 38, 41
 - Markov random field, *see* Markov network
 - maximum likelihood 260
 - MCINTYRE 218, 224, 305, 337, 343, 347
 - mean 27
 - measurable function 25, 47, 48, 96
 - measurable set 23
 - measurable space 23
 - mega-example, *see* mega-interpretation
 - mega-interpretation 271, 296
 - memoing 147, 224
 - Mendelian rules of inheritance 54, 76
 - merge rule 272
 - meta-interpreter 12
 - meta-predicate 65
 - Metropolis-Hasting MCMC 234
 - Metropolis-Hastings MCMC 222, 306
 - mFOIL 284, 291
 - mixed-integer programming 289
 - mode declaration 285, 296
 - model
 - Herbrand 7
 - minimal 8, 12, 111
 - stable 21
 - well-founded 17, 46
 - model counting 151
 - monotonic mapping 3, 17, 104
 - Monte Carlo 213, 216, 337
 - Monty Hall puzzle 52

moralizing a graph 39
 most general unifier 8
 multi-switch, *see* random switch
 multiset 266
 multivalued decision diagrams 155
 multivariate Gaussian density, *see*
 multivariate Gaussian distribution
 multivariate Gaussian distribution
 314
 multivariate normal density, *see*
 multivariate Gaussian distribution

N

natural language processing 309
 necessity 239
 necessity degree 239
 negation as failure 9
 neural network 351
 neutral sum 255
 neutral-sum
 problem 256
 Nilsson's probabilistic logic 84
 noisy-OR 195, 198
 normal approximation 216
 normal density, *see* Gaussian
 distribution
 normal distribution, *see* Gaussian
 distribution
 normal logic program 41
 normalization 38
 normalizing constant 37
 NP-hard 230

O

open sets 110
 open-world assumption 275
 optimal strategy 243

ordinal powers 3
 outcome 23
 output variable 285
 outside probability 262

P

pairwise incompatible set of compos-
 ite
 choices 93
 parameter learning 259, 298
 parameterized interpretation 176
 parameterized negative two-valued
 interpretation 103
 parameterized positive two-valued
 interpretation 102
 parameterized random variable 79,
 196
 parameterized three-valued interpre-
 tation 103
 consistent 103
 parametric factor, *see* parfactor
 parent random variable 34
 parfactor 79, 196
 parfactor graph 195
 parial order 1
 part-of-speech tagging 309
 part-of.speech tagging 311
 partial derivative 169
 partially ordered set 1
 particle filtering 225, 235, 307
 partition function 37
 personalized PageRank 82
 PICL 157
 PIP 187
 PITA 158, 179, 305
 correctness 163
 transformation 159
 PITA(EXC,_) 179
 PITA(IND, EXC) 178

- PITA(IND,EXC) 186
 - PITA(IND,IND) 179
 - PITA(OPT) 181
 - PITA(POSS) 241
 - placemaker 286
 - Poisson distribution 120, 124, 127
 - positive-definite matrix 29
 - possibilistic clause 239
 - possibilistic logic 239
 - possibilistic logic program 240
 - possibilistic theory 239
 - possibility 239
 - possibility degree 239
 - possibility measure 239
 - possible models 47
 - Potassco 22
 - potential, *see* factor
 - powerset 1
 - precision 291
 - precision recall curve 309
 - prediction 33
 - principle of sufficient causation 70
 - principle of universal causation 70
 - PRISM 146, 178, 186, 236, 259, 282
 - language 44, 56
 - learning task 259
 - Probabilistic Constraint Logic
 - Programming 130
 - probabilistic context-free grammar
 - 187, 262, 309, 330
 - probabilistic fact 43, 46
 - Probabilistic Horn Abduction 43, 56
 - probabilistic left corner grammar
 - 309, 310
 - probabilistic logic program 41
 - probabilistic programming 224
 - probability density 26, 234, 308
 - probability distribution 25, 67
 - probability measure 24, 95
 - probability space 24, 101, 126
 - probability theory 41
 - probability tree 66
 - ProbFOIL 290
 - ProbFOIL+ 290
 - learning problem 290
 - ProbLo2 218, 222
 - ProbLog 195, 208, 270
 - language 43–45, 49, 55
 - system 151
 - theory compression 289
 - ProbLog1 213, 217, 243
 - ProbLog2 163, 280
 - parameter learning problem 280
 - product rule 30
 - product space 30
 - product topology 110
 - Progol 82, 284, 298
 - program
 - depth 105
 - temporally well-formed 187
 - project operation 310
 - Prolog 9
 - Prolog Factor Language 79, 197
 - proof procedure 8
 - ProPPR 82
 - Python 175
- Q**
- quantitative structure-activity relationship 304
 - query 8, 33, 46
- R**
- random structure 332
 - random function 236
 - random switch
 - name 44
 - random variable 25, 43, 45

- continuous 25
- discrete 25
- random walk 327
 - one-dimensional 341
- range-restrictedness 5, 300
- recall 285
- recursive clauses 300
- reduction 21
- regression 315
- regular random variable 199
- rejection sampling 222, 234, 306
- relative frequency 260, 266
- relevant
 - ground atom 11, 134
 - rule 11, 164
- reparameterization 37, 268
- resolution 8
 - SLD 8, 164
 - tree 9, 10
 - SLDNF 9, 15
 - tree 9, 15
 - SLG 21
- ROC curve 309

S

- sample space 23
- sampling 213, 216
- SAT 150, 254
- satisfiable set of clauses 7
- saturation 286, 298
- scalar multiplication 245
- schema 285
- selection 46, 49
- selection rule 9
- semantic web 351
- semantics 13
 - Clark's completion 15
 - model theoretic 8
 - stable models 21
 - well-founded 15
- semiring 250
 - commutative 251
 - possibilistic 251
 - probabilistic 251
- sensitivity analysis 254
- sentential decision diagram 173, 234
- set of worlds compatible with a composite choice 93
- set of worlds compatible with a set of composite choices 93
- Shannon expansion 154, 156, 245
- shift operation 310
- sigma-algebra 23, 98
 - minimal 23
 - product 112
- significance test 295
- Skolem normal form 209
- Skolem predicate 208
- Skolemization 208
- SLIPCOVER 296
- Smodels 22
- social network 249
- soft evidence 88
- sound program 46, 71
- soundness 8, 105
- source-to-source transformation 217
- splitting 93
 - algorithm 94, 150
- standard deviation 27
- stick-breaking process 319, 324
- sticker collector problem 341
- stochastic $T_{\mathcal{P}}$ operator 121
- stochastic $T_{\mathcal{P}}$ operator 224
 - cplint 129
- stochastic grammar 81
- stochastic logic program 81, 236, 330
- strategy 242, 335
- stratification 20, 177

- dynamic 17–19
- local 20
- stratum 17
- strictly positive distribution 38
- substitution 6
- success function 189
 - integration 191
 - join 190
 - marginalization 192
 - projection 191
- succinctness 174
- sum rule 31
- summing out, *see* marginalization
- SWI-Prolog 21, 162, 221, 332
- symbolic derivation 187, 282
- symmetric matrix 29

T

- T_P compilation 176, 233
- tabling 21, 148, 158, 161, 164
- target predicate 271, 296
- target predicates 283
- temporal precedence assumption 70
- term 4
- term expansion 217
- text mining 342
- theory compression 289
- theta-subsumption 288
- three-prisoner puzzle 53
- tight logic program 210
- tile map 332
- top element 2, 16, 103
- top-down parsing 310
- top-down reasoning 11
- topic 342
- topological sort 34
- topological space 110
- topology 110
- total composite choice, *see* selection

- transfinite induction 105
- transitive closure 9
- translation
 - BN to LPAD 55
 - LPAD to BN 58
 - LPADs to PHA/ICL 56
 - LPADs to ProbLog 57
 - MLNs to LPADs 85
 - PHA/ICL/PRISM to LPADs 56
- tree
 - SLD 213, 230, 234
- trial id 44
- truel 335
- truth-functional 178
- Tseitin predicate 208
- Turing-complete language 10

U

- uncertainty 23
- undirected graph 38
- unfounded sets 17
- uniform distribution 126
- upper bound 213, 232–234
- upper bound of a set 1
 - least 1
- utility 242
- utility fact 241
- utility function 241
- UW-CSE 301

V

- variable elimination 197
- variance 27
- video game 332
- viral marketing 249
- Viterbi algorithm 150
- Viterbi path 150
- vtree 173

W

weight 37
weight fusion 240
weighted Boolean formula 150, 164,
231
weighted first order model counting
207
weighted MAX-SAT 150, 167, 232
weighted model counting 150, 167,
231
weighted model integration 228

well-defined program 101
well-formed formula 4, 5
world 41–43, 46, 49, 91, 252

X

XSB 21, 162

Y

YAP 21, 78, 162, 218
internal database 218

About the Author

Fabrizio Riguzzi is Associate Professor of Computer Science at the Department of Mathematics and Computer Science of the University of Ferrara. He was previously Assistant Professor at the same university. He got his Master and PhD in Computer Engineering from the University of Bologna.

Fabrizio Riguzzi is vice-president of the Italian Association for Artificial Intelligence and Editor in Chief of *Intelligenza Artificiale*, the official journal of the Association.

He is the author of more than 150 peer reviewed papers in the areas of Machine Learning, Inductive Logic Programming and Statistical Relational Learning. His aim is to develop intelligent systems by combining in novel ways techniques from artificial intelligence, logic and statistics.

FOUNDATIONS OF PROBABILISTIC LOGIC PROGRAMMING

Languages, Semantics, Inference and Learning

Fabrizio Riguzzi

Probabilistic Logic Programming extends Logic Programming by enabling the representation of uncertain information. Probabilistic Logic Programming is at the intersection of two wider research fields: the integration of logic and probability and Probabilistic Programming.

Logic enables the representation of complex relations among entities while probability theory is useful for model uncertainty over attributes and relations. Combining the two is a very active field of study.

Probabilistic Programming extends programming languages with probabilistic primitives that can be used to write complex probabilistic models. Algorithms for the inference and learning tasks are then provided automatically by the system.

Probabilistic Logic programming is at the same time a logic language, with its knowledge representation capabilities, and a Turing complete language, with its computation capabilities, thus providing the best of both worlds.

Since its birth, the field of Probabilistic Logic Programming has seen a steady increase of activity, with many proposals for languages and algorithms for inference and learning.

This book aims at providing an overview of the field with a special emphasis on languages under the Distribution Semantics, one of the most influential approaches. The book presents the main ideas for semantics, inference, and learning and highlights connections among the methods.

Many examples of the book include a link to a page of the web application <http://cplint.eu> where the code can be run online.

ISBN 978-87-7022-018-7



9 788770 220187



River Publishers