ON THE EFFECTIVE REVISION OF (BAYESIAN) LOGIC PROGRAMS

Aline Marins Paes

Doctorate Thesis presented to the Graduate Department of Systems Engineering and Computer Science, COPPE, of Federal University of Rio de Janeiro as a partial fulfillment of the requirements for the degree of Doctor of Systems Engineering and Computer Science.

Advisors: Gerson Zaverucha
Vítor Manuel de Morais Santos Costa

Rio de Janeiro
September, 2011

ON THE EFFECTIVE REVISION OF (BAYESIAN) LOGIC PROGRAMS

Aline Marins Paes

DOCTORATE THESIS PRESENTED TO THE GRADUATE DEPARTMENT OF SYSTEMS ENGINEERING AND COMPUTER SCIENCE, COPPE, OF FEDERAL UNIVERSITY OF RIO DE JANEIRO AS A PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF SYSTEMS ENGINEERING AND COMPUTER SCIENCE.

Committee:

Prof. Gerson Zaverucha, Ph.D.

Prof. Vítor Manuel de Morais Santos Costa, Ph.D.

Prof. Mario Roberto Folhadela Benevides, Ph.D.

Prof. Bianca Zadrozny, Ph.D.

Prof. Stephen Howard Muggleton, Ph.D.

RIO DE JANEIRO, RJ - BRAZIL
SEPTEMBER, 2011

Paes, Aline Marins

On the Effective Revision of (Bayesian) Logic Programs/ Aline Marins Paes. - Rio de Janeiro: UFRJ/COPPE, 2011

XXIV, 282 p.: il.; 29,7 cm.

Advisors: Gerson Zaverucha

Vítor Manuel de Morais Santos Costa

Thesis (PhD) - UFRJ/COPPE/Department of Systems Engineering and Computer Science, 2011

Bibliography: p.240-276

1. Machine Learning. 2. Inductive Logic Programming. 3. Theory Revision from Examples. 4. Probabilistic Logic Learning. 5. Bayesian Networks. 6. Stochastic Local Search. I. Zaverucha, Gerson *et al.*. II. Federal University of Rio de Janeiro, COPPE, Department of Systems Engineering and Computer Science. III. Title.

To my dearest mother Geiza, the pillar of my creation, and to my beloved husband Ricardo, the pillar of my evolution.

Acknowledgments

First, I would like to thank God, for the gift of life. ”God created everything through him, and nothing was created except through him.” (John 1:3)

I could have not completed this thesis without the huge support of my family. I would like to greatly thank them, specially:

I would like to thank my parents, Antonio and Geiza, for everything they denied and sacrificed themselves for providing me the best possible education; for their belief in me, which has been much greater than my belief in myself and for supporting me in all the decisions that I had made that conducted me to this PhD.

I would like to thank my husband Ricardo, for being my greatest encourager, for carefully listening all the ideas and difficulties related to this thesis and for giving so many wise advices; for making my life much more meaningful and happy.

I would like to thank my sister Alessandra and my brother Thiago, my brother-in-law Braz and sister-in-law Marcela, for the affection, care and encouragement.

I would like to thank my niece Susan and my nephew João Vítor, for their ability to fill my life with joy, even in the most stressful times.

And I would like to thank my family of Niteroi, Jane, Ricardo, Janete, Fernando, Arthur, Zélia and Ronald, for all the affection.

I would like to thank all the teachers and professors who have been extremely generous to transmit me knowledge. Specially, I would like to thank the professors that advised and supervised this research:

I thank my advisor, professor Gerson Zaverucha, for all the support, dedication to the research, encouragement and for being a model of determination on pursuing a research. Gerson’s passion for research was a big factor for my decision on being a

researcher. I also thank him for the opportunities he had offered me during the PhD, when contributing and stimulating my participation in prestigious conferences, and for allowing me to co-advise some of his undergraduate students.

I thank my advisor, professor Vítor Santos Costa, for being always available to help me on my numerous doubts. I thank him very much for his generosity on the development, maintenance and continuous update of YAP Prolog, the best Prolog compiler ever. I thank him for all the valuable tips of how to elegantly write English. Additionally, I would like to thank him and his family for hosting me so nicely in their home in Portugal.

I thank my internship supervisor, professor Stephen Muggleton, for allowing me to be part of the *Computational Bioinformatics Laboratory* of Imperial College London for a year. His generosity, creativity, patience and so profound scientific knowledge have greatly contributed to this research and to my own development as a researcher.

I would like to thank my friends for the great support they have provided me along these years, specially:

Kate Revoredo, for being a model of determination, for the partnership, collaboration and a lot of great advices.

Ana Luísa Duboc, for the partnership and encouragement.

Past and present friends of the AI Lab, Juliana Bernardes, Cristiano Pitangui, Carina Lopes, Elias Barenboim, Fábio Vieira, Glauber Menezes, Aloísio Pina, for sharing ideas over these years and for providing a fun workplace.

Friends of the Computational Bioinformatics Laboratory, José Santos, Ramin Ramezani, Pedro Torres, Niels Pahlavi, Jianzhong Chen, Alireza Tamaddoni-Nezhad, Hiroaki Watanabe, Robin Baumgarten and John Charnley, for sharing ideas during the year I have worked in Imperial College and for being such wonderful hosts.

My undergraduate "co-students", Eric Couto, Rafael Pereira and Guilherme Niedu for their collaboration.

All my friends, those far away and those close to me. I know that they have always kept me on their good thoughts.

I thank very much the committee, professors Stephen Muggleton, Mário Bene-

vides and Bianca Zadrozny, for the careful analysis of so many pages in such little time and for the greatly valuable comments made on this thesis.

I thank the PESC secretaries, Claudia, Solange, Sônia and Gutierrez, and CBL secretary, Bridget, who are always available to solve any administrative issue.

I thank the PESC support team, Itamar, Adilson, João Vítor, Alexandre and Thiago, who are always ready to solve any physical and network issues.

I thank UFRJ and Imperial College London for the physical installations.

And finally I thank CAPES and CNPq for the financial support.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.SC.)

Aline Marins Paes

September/2011

Advisors: Gerson Zaverucha
Vítor Manuel de Morais Santos Costa

Department: Systems Engineering anc Computer Science

Theory Revision from Examples is the process of improving user-defined or automatically generated theories, guided by a set of examples. Despite several successful theory revision systems having been built in the past, they have not been widely used. We claim that the main reason is the inefficiency of these systems, as they usually yield large search spaces. This thesis contributes towards the goal of designing feasible theory revision systems. First, we focus on first-order theory revision. We introduce techniques from Inductive Logic Programming (ILP) and from Stochastic Local Search to reduce the size of each individual search space generated by a FOL theory revision system. We show through experiments that it is possible to have a revision system as efficient as a standard ILP system and still generate more accurate theories. Moreover, we present an application involving the game of Chess that is successfully solved by theory revision, in contrast with a learning from scratch system that fails in correctly achieving the required theory. As first-order logic handles well multi-relational domains but fails on representing uncertain data, there is a great recent interest in joining relational representations with probabilistic reasoning mechanisms. We have contributed with a probabilistic first-order theory revision system called PFORTE. However, despite promising results in artificial domains, PFORTE faces the complexity of searching and performing probabilistic inference over large search spaces, making it not feasible to be applied to real world domains. Thus, the second major contribution of this thesis is to address the bottlenecks of probabilistic logic revision process. We aggregate techniques from ILP and probabilistic graphical models to reduce the search space of the revision process and also of the probabilistic inference. The new probabilistic revision system was successfully applied in real world datasets.

Contents

Acknowledgments v

Nomenclature xiii

List of Algorithms xv

List of Figures xviii

List of Tables xxiv

1 Introduction 1

1.1 First-order Logic Theory Revision from Examples 2

1.1.1 Contributions to First-order Logic Theory Revision 4

1.2 Probabilistic Logic Learning 7

1.2.1 BFORTE: Towards a Feasible Probabilistic Revision System . 8

1.3 Publications 11

1.4 Thesis outline 12

2 ILP and Theory Revision 14

2.1 Inductive Logic Programming 15

2.1.1 Learning Settings in ILP 17

2.1.2 Ordering the Hypothesis Search Space 18

2.1.3 Mode Directed Inverse Entailment and the Bottom Clause . . 18

2.2 First-order Logic Theory Revision from Examples 23

2.2.1 Revision Points 25

2.2.2 Revision Operators 26

2.2.3 FORTE 27

3 YAVFORTE: A Revised Version of FORTE, Including Mode Directed Inverse Entailment and the Bottom Clause 42
3.1 Introduction 42
3.2 Restricting the Search Space of Revision Operators 44
3.3 Improvements Performed on the Revision Operators 46
3.3.1 Using the Bottom Clause as the Search Space of Antecedents when Revising a FOL theory 46
3.3.2 Modifying the Delete Antecedent Operator to use Modes Language and to Allow Noise 53
3.4 Experimental Results 54
3.5 Conclusions 63

4 Chess Revision: Acquiring the Rules of Variants of Chess through First-order Theory Revision from Examples 66
4.1 Motivation 66
4.2 Revision Framework for Revising Rules of Chess to Turn Them in the Rules of Variants 70
4.2.1 The Format of Chess Examples 70
4.2.2 The Background Knowledge 75
4.3 Modifying YAVFORTE to Acquire Rules of Chess Variants 80
4.3.1 Starting the Revision Process by Deleting Rules 80
4.3.2 Using abduction during the revision process 81
4.3.3 Using negated literals in the theory 89
4.4 Experimental Results 92
4.4.1 Discussions about the automatic revisions performed by the revision system 94
4.5 Conclusions 98

5 Stochastic Local Search 100
5.1 Stochastic Local Search Methods 101
5.1.1 SLS Methods Allowing Worsening Steps 102
5.1.2 Rapid Random Restarts 104
5.1.3 Other SLS approaches 105
5.2 Stochastic Local Search Algorithms for Satisfiability Checking of Propositional Formulae 105
5.2.1 The GSAT Algorithm 106
5.2.2 The WalkSAT Algorithm 107
5.3 Stochastic Local Search in ILP 109

5.3.1 Stochastic Local Search for Searching the Space of Individual Clauses . . . 110
5.3.2 Stochastic Local Search in ILP for Searching the Space of Theories and/or using Propositionalization . . . 112

6 Revising First-Order Logic Theories through Stochastic Local Search 117
6.1 Introduction . . . 117
6.2 Stochastic Local Search for Revision Points . . . 119
6.3 Stochastic Local Search for Literals . . . 122
6.3.1 Stochastic Component for Searching Literals . . . 124
6.4 Stochastic Local Search for Revisions . . . 131
6.4.1 Stochastic Greedy Search for Revisions with Random Walk . . . 132
6.4.2 Stochastic Hill-Climbing Search for Revisions with Random Walks . . . 136
6.4.3 Hill-Climbing Search for Revisions with Stochastic Escapes . . . 139
6.4.4 Simulated Annealing Search for Revisions . . . 140
6.5 Experimental Results . . . 143
6.5.1 Datasets . . . 145
6.5.2 Behavior of the Stochastic Local Search Algorithms with Different Parameters . . . 146
6.5.3 Comparing Runtime and Accuracy of SLS Algorithms to Aleph and YAVFORTE . . . 151
6.6 Conclusions . . . 155

7 Probabilistic Logic Learning 159
7.1 Bayesian Networks: Key Concepts . . . 161
7.1.1 D-separation . . . 164
7.1.2 Inference in Bayesian Networks . . . 172
7.1.3 Learning Bayesian Networks from Data . . . 172
7.2 Bayesian Logic Programs . . . 177
7.2.1 Bayesian Logic Programs: Key concepts . . . 179
7.2.2 Answering queries procedure . . . 181
7.2.3 Learning BLPs . . . 184
7.3 PFORTE: Revising BLPs from Examples . . . 190
7.3.1 PFORTE: Key concepts . . . 191
7.3.2 The PFORTE Algorithm . . . 193

8 BFORTE: Addressing Bottlenecks of Bayesian Logic Programs Revision 200

8.1 Addressing Search Space of New Literals . . . 201
8.2 Addressing Selection of Revision Points . . . 203
8.2.1 Bayesian Revision Point . . . 203
8.2.2 Searching Bayesian Revision Points through D-Separation . . 205
8.2.3 Analyzing the Benefits Brought by Revision Operators According to D-Separation . . . 207
8.3 Addressing Inference Space . . . 219
8.3.1 Collecting the Requisite Nodes by d-separation . . . 220
8.3.2 Grouping together Ground Clauses and Networks . . . 221
8.4 Experimental results . . . 223
8.4.1 Comparing BFORTE to ILP and FOL Theory revision . . . 226
8.4.2 Speed up in the revision process due to the Bottom clause . . 227
8.4.3 Selection of Revision Points . . . 228
8.4.4 Inference time . . . 230
8.5 Conclusions . . . 232

9 General Conclusions and Future work 235

9.1 Summary . . . 235
9.2 Future work: First-Order Logic Theory Revision . . . 237
9.3 Future work and work in progress: Probabilistic Logic Revision . . . 238
9.3.1 Revision of Bayesian Logic Programs . . . 238
9.3.2 Revision of Stochastic Logic Programs . . . 238

Bibliography 240

Index 276

A The Laws of International Chess 279

Nomenclature

AIC Akaike's Information Criteria

BLP Bayesian Logic Program

CLP(BN) Constraint Logic Programming with Bayes Nets

CLL Conditional Log-likelihood

CPD Conditional Probability Distribution

DAG Directed Acyclic Graph

EM Expectation-Maximization

FN False Negative

FORTE First-Order Revision of Theories from Examples

FP False Positive

ICI Independence of Causal Influence

ICL Independent Choice Logic

IID Independently and Identically Distributed

ILS Iterated Local Search

ILP Inductive Logic Programming

LL Log-Likelihood

LPM Logical Probabilistic Model

MDIE Mode Directed Inverse Entailment

MDL Minimum Description Length

MLE Maximum Likelihood Estimator

MLN Markov Logic Networks

PCFG Probabilistic Context-free Grammar

PGM Probabilistic Graphical Model

PILP Probabilistic Inductive Logic Programming

PII Probabilistic Iterative Improvement

PLL Probabilistic Logic Learning

PLM Probabilistic Logical Model

PRM Probabilistic Relational Models

RII Randomised Iterative Improvement

RRR Rapid Random Restarts

SA Simulated Annealing

SLP Stochastic Logic Program

SLS Stochastic Local Search

SRL Statistical Relational Learning

TN True Negative

TP True Positive

wp Walk Probability

List of Algorithms

2.1 Top-level Algorithm of the Bottom Clause Construction Process (De Raedt, 2008) . . . 19
2.2 Bottom clause construction Algorithm (Muggleton, 1995) . . . 21
2.3 FORTE Algorithm (Richards and Mooney, 1995) . . . 28
2.4 FORTE Algorithm for collecting specialization revision points (Richards and Mooney, 1995) . . . 30
2.5 FORTE Algorithm for collecting generalization revision points (Richards and Mooney, 1995) . . . 32
2.6 FORTE Delete Rule Revision Operator Algorithm . . . 33
2.7 FORTE Top Level Add Antecedents Revision Operator Algorithm . . . 33
2.8 Hill climbing add antecedents Algorithm . . . 34
2.9 Top level Algorithm for Adding Antecedents to a Clause Using Relational Pathfinding Approach . . . 36
2.10 Hill Climbing Antecedents Generation Algorithm . . . 37
2.11 Relational Pathfinding Antecedent Generation Algorithm (Richards and Mooney, 1995) . . . 38
2.12 Top-level Delete Antecedents Revision Operator Algorithm . . . 39
2.13 Hill Climbing Delete Antecedents Algorithm . . . 40
2.14 Delete Multiple Antecedents Algorithm . . . 40
2.15 Top-level Add rule Revision Operator Algorithm . . . 41
3.1 YAVFORTE Top-Level Algorithm . . . 45
3.2 Bottom clause Construction Algorithm in FORTE-MBC . . . 49
3.3 Hill Climbing Add Antecedents Algorithm Using the Bottom Clause . . . 50
3.4 Top-level Relational Pathfinding Add Antecedents Algorithm Using the Bottom Clause . . . 51
3.5 Remodeled Greedy Hill Climbing Delete Antecedents Algorithm . . . 55
4.1 Chess examples generator Algorithm . . . 75
4.2 First Step for Deleting Rules . . . 81

4.3 Algorithm for Refining a Clause whose Head Corresponds to an Intermediate Predicate . . . 85
4.4 Algorithm for fabricating an intermediate instance . . . 85
4.5 Atomic Facts Abduction Algorithm . . . 87
5.1 Top-level Step Function Performed by an Algorithm using Randomised Iterative Improvement method (Hoos and Stützle, 2005) . . . 103
5.2 Simulated Annealing algorithm (Geman and Geman, 1984) . . . 104
5.3 GSAT Algorithm (Selman et al., 1992) . . . 107
5.4 WalkSAT General Algorithm (Selman et al., 1994) . . . 108
5.5 Most Used Heuristic Function into WalkSAT Architecture . . . 109
5.6 A SLS algorithm to learn k-term DNF formulae (Rückert and Kramer, 2003) . . . 115
6.1 SLS Algorithm for generating revision points . . . 121
6.2 Algorithm for adding antecedents using hill-climbing SLS . . . 128
6.3 Stochastic Relational-pathfinding . . . 129
6.4 Algorithm for deleting antecedents using hill-climbing SLS . . . 130
6.5 Stochastic version of the algorithm for deleting multiple antecedents . 131
6.6 Stochastic Greedy Search for Revisions with Random Walk . . . 135
6.7 Stochastic Hill-climbing Search for Revisions with Random Walk . . . 138
6.8 Hill-Climbing Search for Revision with Stochastic Escape . . . 141
6.9 Simulated Annealing Search for Revisions . . . 144
7.1 The Bayes Ball Algorithm to Collect Requisite Nodes in a Network (Shachter, 1998) . . . 170
7.2 Algorithm for Inducing a Support Network from a Probabilistic Query (Kersting and De Raedt, 2001a) . . . 182
7.3 Algorithm for learning BLPs (Kersting and De Raedt, 2001b) . . . 190
7.4 PFORTE Algorithm (Paes et al., 2006a) . . . 195
7.5 Algorithm for generating Probabilistic Revision Points in a Logical Probabilistic Model such as BLP . . . 196
7.6 Algorithm for deletion/addition of antecedents in PFORTE . . . 197
7.7 Algorithm to Score Possible Revisions in PFORTE . . . 199
8.1 BFORTE Top-Level Algorithm . . . 205
8.2 Top-level Selection of Revision Points . . . 207
8.3 Generation of Revision Points and Revisions Top-Level Procedure . . 219
8.4 Top-level Procedure for Performing Inference over Requisite Nodes Only . . . 221
8.5 Top-level Procedure for Detecting Similar Ground Clauses and Mapping them to Only One . . . 223

8.6 Top-level Procedure for Detecting Similar Networks and Mapping them to Only One 224

List of Figures

2.1 Schema of the learning task in ILP, where BK is background knowledge, E is the set of positive (E^+) and negative (E^-) examples and H' is the theory learned by the ILP system. 16

2.2 Schema of Theory Revision from Examples, where FDT is the fixed preliminary knowledge and H' is the modifiable preliminary knowledge, E is the set of positive (E^+) and negative (E^-) examples. H is the theory returned by the revision system 24

2.3 An instance of a relational graph representing part of the family domain (Richards and Mooney, 1995) 35

3.1 Scatter plot for all datasets and systems settings considering theories learned by Aleph with its default parameters 62

3.2 Scatter plot for all datasets and systems settings considering theories learned by Aleph with its default parameters, except for clause length 62

3.3 Scatter plot for all datasets and systems settings considering theories learned by Aleph with better parameters settings than just default . . 63

4.1 Visualization of situations when an ILP system should learn definitions for subconcepts. Figure (a) shows a board of chess with a checked king. Figure (b) shows a piece working as a pin. 67

4.2 Boards of variants of chess. In the first row it is the international Chess, on its side it is the Xiangqi (Chinesse chess), followed by Shogi (Japanese Chess). Next row it one of the first chess games, the Chaturanga in the Hindu version, followed by the Chess in the Round game and a more modern version of Chaturanga, quite close to the International Chess. In the last group it is Shogi, Anti King Chess, Los Alamos and Grand chess, in this order. 69

4.3 Components of the chess revision framework: Initial theory is going to be revised by the Revision system, using FDT and Dataset. Dataset is created from the Examples generator component. 71
4.4 The initial setting of the board in international chess and atomic facts corresponding to it. Figure of the board is taken from $http : //www.mark - weeks.com/aboutcom/bloa0000.htm$ 73
4.5 Initial boards of minichess games, taken from Wikipedia 93

6.1 Comparing runtime and accuracy of Algorithm 6.1 in Pyrimidines Dataset, with number of revision points varying in 1, 5, 10, 20. Probabilities are fixed in 100% and 50%. 147
6.2 Comparing runtime and accuracy of Algorithm 6.1 in Proteins Dataset with number of revision points varying in 1, 5, 10, 20. Probabilities are fixed in 100% and 50%. 147
6.3 Comparing runtime and accuracy of SLS algorithms in Pyrimidines Dataset, varying maximum number of iterations, which is set to 10, 20, 30, 40. Probabilities are fixed in 100% and 50%, when it is the case. 148
6.4 Comparing runtime and accuracy of SLS algorithms in the proteins dataset, varying maximum number of iterations, which is set to 10, 20, 30, 40. Probabilities are fixed in 100% and 50%, when it is the case. 149
6.5 Comparing runtime and accuracy of SLS algorithms in Pyrimidines Dataset, with probabilities varying in 100, 80, 60, 40. Maximum number of iterations is fixed in 20, when it is the case. 150
6.6 Comparing runtime and accuracy of SLS algorithms in Proteins Dataset, with probabilities varying in 100, 80, 60, 40. Maximum number of iterations is fixed in 20, when it is the case. 150
6.7 Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Pyrimidines Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 153
6.8 Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Proteins Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 153

6.9 Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Yeast Sensitivity Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 154
6.10 Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Pyrimidines Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 155
6.11 Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Proteins Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 155
6.12 Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Yeast Sensitivity Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis. 156

7.1 Bayesian network representing the blood type domain within a particular family 164
7.2 The four possible edge trails from X to Y, via E. Figure (a) is an indirect causal effect; Figure (b) is an indirect evidential effect; Figure (c) represents a common cause between two nodes and Figure (d) shows a common effect trail. 167
7.3 The ball visiting nodes with Bayes Ball algorithm. Red nodes are observed and green node is the query node. 171
7.4 Support networks created from the query variable $bt(susan)$ and evidence variables $bt(brian)$, $pc(brian)$, $pc(susan)$, $pc(allen)$ and $bt(allen)$. 183
7.5 Bayesian network after joining same random variables from different support networks from Figure 7.4. 184
7.6 Bayesian network representing the blood type domain within a particular family. Nodes in magenta are yielded by the same Bayesian clause. 186

7.7 Figure (a) is a Bayesian network without adding extra nodes to represent decomposable combining rules. Node $pred(1)$ is produced by three different clauses. Figure (b) is the induced network reprsenting decomposable combining rules by adding extra nodes (yellow nodes) so that each node is produced by exactly one Bayesian clause. Nodes $n_?pred(1)$ has the domain of $pred$ and cpd(c) associated. 188
7.8 Decomposable combining rules expressed within a support network. Instantiations of the same clause and different clauses are combined separately. Nodes $l1_n?_pred(1)$ represent different instantiations of the same clause and nodes $l2_n?_pred(1)$ represent different clauses. . 189

8.1 Non-observed node visited from a child, where shadow nodes are observed nodes. Figure (a) shows a non-observed node visited from a child. Figure (b) shows this node passing the received ball to its parents and children. 209
8.2 An example of the effect of deleting a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario. 209
8.3 An example of the effect of adding antecedentes to a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario. 210
8.4 The effect of deleting antecedentes to a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario 210
8.5 The effect of adding a new rule from a non-observed node visited from a child, in a misclassification node visiting scenario. Figure (a) shows a situation where the old and new rules had to be combined. Figure (b) shows the case where both rules were not combined. 211
8.6 Observed node visited from a child, where shadow nodes are observed nodes. Figure (a) shows an observed node visited from a child. Figure (b) shows this node blocking the ball. 214
8.7 Non-observed node visited from a parent, where shadow nodes are observed nodes. Figure (a) shows a non-observed node visited from a parent. Figure (b) shows this node passing the ball to its children. . 215

8.8 The effect of deleting antecedents from a clause whose head is a non-observerd node visited from a parent. Figure (a) is the network before deleting antecedents. Red node is the visiting parent and yellow node is the visited node. Figure (b) shows a possible resulting network, after the clause becomes more general. 216
8.9 Observed node visited from a parent, where shadow nodes are observed nodes. Figure (a) shows a observed node visited from a parent. Figure (b) shows this node passing the ball back to its parents. 217

A.1 Initial position of a Chess board: first row: rook, knight, bishop, queen, king, bishop, knight, and rook; second row: pawns 280
A.2 Castling from the queen side and from the king side (figure is due to *http : //www.pressmantoy.com/instructions/instruct_chess.html*) . 281
A.3 En-passant move of a pawn in the Game of Chess. Figure is from *http://www.learnthat.com/courses/fun/chess/beginrules15.shtml* . . . 282

List of Tables

2.1 Some standard Logic Programming terms and their definitions 15

3.1 Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer amine dataset 58
3.2 Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer toxic dataset 59
3.3 Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer choline dataset 59
3.4 Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer Scopolamine dataset 60
3.5 Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for DssTox dataset 60

4.1 Format of one example in Chess dataset 72
4.2 Part of one example in Mini-chess Dataset 74
4.3 Ground facts representing pieces and colors in the game and file and ranks of the board, considering the interantional Chess 77
4.4 An extracted piece of the chess theory, to exemplify the need of abduction when learning intermediate concepts 82
4.5 An extracted piece of the chess theory, to exemplify the need of abducting predicates when searching for revision points 88
4.6 An extracted piece of the chess theory, to exemplify the need of marking a negated literal as a revision point 92
4.7 Evaluation of the revision on chess variants 94

7.1 Propositional Logic Program representing the network in Figure 7.1 . 178
7.2 First-order Logic Program representing the regularities in the network of Figure 7.1 178

7.3 First-order Logic Program representing the regularities in the network of Figure 7.1, where each CPD is represented as a list of probability values. 179
7.4 Qualitative part of a Bayesian Logic Program 182
7.5 Format of an example in PFORTE system 192

8.1 BFORTE compared to Aleph and FORTE 226
8.2 Comparison of runtime and accuracy of BFORTE with Bottom Clause and BFORTE using FOIL to generate literals. 228
8.3 Comparison of BFORTE runtime and score with PFORTE and BLP algorithms for selecting points to me modified. 230
8.4 Inference runtime in seconds for UW-CSE dataset. 231
8.5 Inference runtime in seconds for Metabolism dataset. 231
8.6 Inference runtime in seconds for Carcinogenesis dataset. 232

Chapter 1

Introduction

Artificial Intelligence is concerned with building computer programs that solve problems which would require intelligence if solved by a human. As intelligence requires learning, to build computer programs that can learn plays a central role in artificial intelligence. This task is the subject of the area of machine learning, whose ultimate goal is to construct computer programs that can automatically improve their behavior with experience (Mitchell, 1997). Traditional machine learning algorithms learn from independent homogeneous examples, described in attribute-value (or propositional) format. However, in many real world applications, data are multi-relational, highly structured and sampled from complex relations. Consequently, propositional algorithms are not appropriate to learn from them. In this case, formalisms such as first-order logic are more adequate to represent such data than the classical propositional format.

Inductive Logic Programming (Muggleton, 1992; Muggleton and De Raedt, 1994) is the process of automatically learning First-Order Logic Theories from a set of examples and a fixed body of prior knowledge, the *background knowledge*, both written as logical clauses. ILP offers several advantages. It learns from an expressive language; it is easy for humans to understand; the background knowledge is a useful tool to guide the learning process. A large number of algorithms and systems have been developed towards learning in this context. Popular examples include FOIL (Quinlan, 1990), Progol (Muggleton, 1995), Claudien (De Raedt, 1997), Aleph (Srinivasan, 2001b), Tilde (Blockeel and De Raedt, 1998), among many others. ILP has been successfully applied on a number of applications, mainly involving

chimio and bio-informatics (Muggleton et al., 1992; Bratko and King, 1994; Srinivasan et al., 1997; Srinivasan et al., 2006; Tamaddoni-Nezhad et al., 2006; Srinivasan and King, 2008). Most ILP systems learn one logical clause at each time, employing a covering approach: after learning a clause, positive examples covered by it are removed from the set of examples, so that only unprovable positive examples remain to be explained by new clauses.

1.1 First-order Logic Theory Revision from Examples

ILP systems consider background knowledge as correct. However, it may be the case that prior knowledge is available but it is incomplete or only partially correct. The background theory may have been elicited from a domain expert, who relies on incorrect assumptions or who only has partial, even if useful, understanding of a domain. Or it may be that new examples, that cannot be explained by the current theory, have become available. Still, there may the case that a theory has been learned/eilicited for a domain and one would like to transfer it to a related domain. In all such cases, since the initial theory probably contains important information, one would like to take advantage of the original theory as a starting point for the learning process, and repair it or improve it. Ideally, this should accelerate the learning process and result in more accurate theories.

Several *theory refinement* systems have been proposed towards this goal (Shapiro, 1981; Muggleton, 1987; Wogulis and Pazzani, 1993; Wrobel, 1994; Adé et al., 1994; Wrobel, 1996; Richards and Mooney, 1995; Garcez and Zaverucha, 1999; Esposito et al., 2000). Such systems assume the initial theory is approximately correct. If so, then only some *points* (clauses and/or literals) in the theory prevent it from correctly modeling the dataset. Therefore it should be more effective to search for such points in the theory and *revise* them, than to use an algorithm that learns a whole new theory from scratch.

Thus, theory revision systems operate by searching for *revision points*, which are the points considered as responsible for the faults in the theory, and then proposing *revisions* to such points, through applying at each one a number of *revision operators*. Theory revision systems thus learn from whole theories, instead of indi-

vidual clauses. Despite the advantages of refining complete theories (Bratko, 1999), this unfortunately produces a large search space of proposed hypothesis.

Consider, for example, the FORTE (Richards and Mooney, 1995) system, that automatically revises first-order logic theories from a set of positive and negative examples. Its first step is to search for points responsible for misclassifying positive and negative examples. After identifying all revision points in the current theory, FORTE proposes revisions to each one of them, using a set of revision operators. Those operators include adding/deleting antecedents to/from a clause and adding/deleting rules to/from the theory. Finally, from the set of proposed revisions, FORTE chooses one to be implemented and restarts the cycle. Arguably, each step of such a revision process produces a large search space, depending on the number of revision points and the amount of possible revisions that can be proposed to each one of them.

Thus, despite the various theory revision systems developed in the past (Wrobel, 1996) they are not widely used anymore. There are two main reasons for that, as pointed out in (Dietterich et al., 2008): (1) the lack of applications with substantial codified initial theory and (2) as discussed above, the large search space theory revision systems explore. The first problem does not hold any longer, as large-scale resources of background knowledge in areas such as biology have become available (Muggleton, 2005). Therefore, there is an increasing need for efficient theory revision systems, that can fulfill the promise of theory revision.

The work in this thesis contributes toward the goal of obtaining more efficient relational revision systems, both for purely logical and for probabilistic relational theories. We demonstrate that search can be substantially improved by techniques such as bottom-clause based search or stochastic search, with comparable or improved performance, and we demonstrate that the techniques allow revising challenging applications. Next, we introduce each individual contribution of the present thesis.

1.1.1 Contributions to First-order Logic Theory Revision

In this work we focus on FORTE system, since it revises theories automatically from a set of positive and negative examples. FORTE performs an iterative hill climbing search in three steps. First, it searches for the points in the current theory considered as responsible for the misclassification of some example. Second, it searches for modifications at each revision point considering appropriate revision operators, so that a number of revisions to the theory are proposed. Third, it must choose which revision is going to be implemented.

Execution time strongly depends on the second step. When searching for modifications to be implemented in the theory, FORTE takes into account operators that attempt to generalize or specialize the theory, according to the revision point. To specialize theories, it tries to simply *delete rules* and/or *add antecedents* to existing rules. To generalize theories, it may *delete antecedents* from existing rules or *add new rules* to the theory[1]. Depending upon the number of revision points, FORTE may take a large amount of time to employ each revision operator to each revision point. Additionally, it may be the case that the expert of the domain already has some idea or constraint about how the theory can be modified. For example, he/she could want a new theory that would preserve all the old clauses and therefore *deletion of rules* would not be allowed. Considering those issues, we modified the FORTE algorithm so that not all operators are employed to propose modifications to the theory. First, we observe that it is possible that a simpler operator (for example, *delete rules* compared to *add antecedents*) has already achieved the goal of the revision point and therefore it would not be necessary to propose modifications with a more complex operator. Second, we allow the expert to decide beforehand which operator(s) is going to be employed in the revision process.

We further observe that FORTE spends a large amount of time choosing literals to be added to clauses, namely when applying the add antecedents and add rules operator. This is due to the top-down approach borrowed from FOIL (Quinlan, 1990). In this approach, literals are generated from the knowledge base, without

[1]FORTE has others operators that we do not consider in this work, such as absorption and identification

taking into account the set of examples. The dataset is employed only to score the literals so that the best one is chosen to be added to a clause. On the other hand, standard ILP algorithms such as Progol (Muggleton, 1995) take advantage of a hybrid bottom-up and top-down approach to refine clauses. First, they create a *Bottom Clause* from a single positive example, following a Mode Directed Inverse Entailments (MDIEs) approach (Muggleton, 1995). Next, this variabilized Bottom Clause composes the search space for possible antecedents to be added to a clause. Arguably, the size of the search space can be greatly reduced, when compared to trying all possible antecedents generated from the knowledge base. Thus, a second contribution of this thesis is to replace the FOIL literals generation approach by the use of the Bottom clause. Sets of literals subsuming the Bottom Clause are scored and the best one is chosen to be added to a clause. The system built upon the modifications discussed in the last two paragraphs is called here YAVFORTE and it includes FORTE_MBC (Duboc et al., 2009) as the algorithm for collecting literals from the Bottom clause.

Next, we have designed a challenging application to show the power of revision compared to learning from scratch. The application concerns the revision of the game of Chess to acquire the rules of variants of this game. Throughout the years the game of chess has inspired several variants, either to be more challenging to the player, or to produce an easier and faster variant of the original game (Pritchard, 2007). It also has several different regional versions, such as the Chinese and Japanese versions. Ideally, if the rules of the chess have been obtained, we would like to use them as a starting point to obtain the rules of a variant. However, such rules need non-trivial changes in order to represent the particular aspects of the variant. In a game such as chess this is a complex task that may require addressing different board sizes, introducing or deleting new promotion and capture rules, and may require redefining the role of specific pieces in the game. Thus, we address this problem as an instance of theory revision from examples. Additionally, we show that for effectively tackling this problem, the revision system handles *abduction* (Flach and Kakas, 2000a) and *negation as failure* (Clark, 1978). In this way, these both thecniques are integrated to the revision process of YAVFORTE system.

Although those earlier developments are capable of improving the runtime of the revision process without harming the accuracy of the system, the performance of the revision system must still be improved when it faces large initial theories and datasets. When looking for revision points, the system must first select the misclassified examples. Next, it must mark the clauses responsible for such misclassification. In case the initial theory is composed of a large amount of clauses and/or there are several (misclassified) examples, the search for revision points can be very expensive. Moreover, if after selecting the revision points there is a large amount of clauses marked to be revised, several possible modifications are going to be proposed. Additionally, it may be the case that the background knowledge and the language of the theory (predicates and modes declarations) is large enough to produce a huge bottom clause. In those situations, the modifications we described before may not lead to a acceptable revision time. In order to overcome the intractability of the revision process in those situations, we propose to abandon completeness in favor of finding good solutions in a reasonable amount of time. To achieve this goal we make use of *stochastic local search techniques* (Hoos and Stützle, 2005) to introduce randomization into to the searches performed during the revision process. We include stochastic search components in the key searches within the revision process of YAVFORTE: the search for revision points(1), the search for literals to be added/deleted to/from a clause when proposing modifications within a revision operator(2) and the revision operator that is going to indeed modify the theory(3).

To summarize, the first major contribution of this thesis, concerning first-order logic revision, is the development of a effective revision system named YAVFORTE. YAVFORTE has reduced search space of new literals, revision points and revision operators compared to FORTE. The spaces are limited mainly by the use of the Bottom Clause and stochastic components. The revision process is stronger also by the use of stochastic search and by the integration of abduction and negation as failure in the revision process.

1.2 Probabilistic Logic Learning

The successful applications of classical ILP to relational domains is often limited by the need of representing uncertainty and partially observed information. Although traditional statistical machine learning techniques have the ability to handle uncertainty, they cannot represent relational domains, as they are essentially propositional. Thus, recently there has been a great interest in the integration of logical and probabilistic reasoning mechanisms. Building languages and algorithms for learning programs in these languages is the main subject of a new area of Artificial Intelligence named Probabilistic Logic Learning (PLL), also known as Probabilistic Inductive Logic Programming (PILP) and quite related to Statistical Relational Learning (SRL). PLL deals with machine learning in relational domains where information may be missed, only partially observed, noisy or uncertain. Several formalisms have been developed in this area in the last decades, including SLP (Stochastic Logic Programs) (Muggleton, 1996), PRM (Probabilistic Relational Models) (Koller and Pfeffer, 1998), BLP (Bayesian Logic Programs) (Kersting and De Raedt, 2001b), CLP(BN) (Constraint Logic Programming with Bayes net) (Santos Costa et al., 2003a), MLN (Markov Logic Networks) (Richardson and Domingos, 2006), ProbLog (De Raedt et al., 2007), among many others. In most such systems, knowledge is represented by definite clauses annotated with probability distributions. Inference may be performed either using a logical mechanism taking into account the probabilities, or through a probabilistic graphical model, built from the examples and the current model. This is, for example, the case of BLPs, that generalize both logic programs and Bayesian networks. Inference is performed over Bayesian networks, built from each example and clauses in BLP through a Knowledge Base Model Construction (KBMC) approach (Ngo and Haddawy, 1997).

Most algorithms developed in PLL assume either the model must be learned from scratch, from background knowledge and dataset, or the rules must be modified as a whole, considering they are in the same level of correctness. This last case is the approach followed by BLPs learning algorithm. There is less work on the task of *repairing* or *improving* an initial probabilistic logic model. As examples of models that could be fixed/improved are the ones elicited by an expert of the domain,

containing useful information but that not guaranteed to reflect the set of examples. Also, it may be the case that an initial first-order theory was learned from an ILP system and therefore noise and uncertainty were not taken into account during the learning process.

1.2.1 BFORTE: Towards a Feasible Probabilistic Revision System

As the learning task is time consuming and the initial model may contain valuable information, one would like to consider it as a starting point and *refine* it. Ideally, this should result in faster learning time and more accurate models. Motivated by the benefits brought by first-order theory revision to the learning task, we have proposed to automatically *revise* an initial Bayesian Logic Program in (Revoredo and Zaverucha, 2002; Paes et al., 2005b; Paes et al., 2005a; Paes et al., 2006a), resulting in a system we called PFORTE, since it is based on FORTE.

Learning or revising probabilistic first-order theories does introduce interesting novel issues. In PLL it is convenient to see examples as evidence for random variables. The distribution of probabilities will then give an expectation for whether an example will take a specific value, say the true value or the false value. Following a discriminative approach, a theory should be revised if it fails on generating the appropriate probability distribution for an example. In this case, we say the example is misclassified. PFORTE's strategy is as follows. First, it addresses theory incompleteness, by finding the points that failed when proving examples. PFORTE uses generalization operators to propose modifications to those points, choosing the best one to correct through a scoring function. This phase uses a hill-climbing search and stops when we cannot improve example covering. Second, PFORTE uses the generalized theory as starting point for search in the space composed by generalization and specialization on points of the theory that failed in producing a proper probability distribution. In this case, clauses taking part in the Bayesian network model generated from a misclassified example are considered as revision points. After this step we expect a theory more accurate in classification.

Although with PFORTE we experimentally demonstrated that it is possible to

obtain more accurate models compared to learning from scratch, the time it spends to achieve that is prohibitive. We have identified at least three bottlenecks of the revision process:

- Choice of the revision points: As stated before, the BLP learning algorithm starts from an initial most general theory and proposes modifications to each clause, in an attempt to represent the uncertainty in the model. PFORTE, on the other hand, proposes modification to every clause used to construct the Bayesian network where an example is misclassified. Albeit PFORTE reduces the number of clauses subject to modifications compared to BLP learning, it may still select several clauses to be revised. The problem is, as the Bayesian network is built from a relational example, composed of several ground facts, it usually contains several clauses from the BLP, with many of them not even *relevant* to compute the probability distribution of an instance.

- Inference time: Bayesian networks built from relational examples must capture the relationships between different objects of the domain. This fact, added to the inherent complexity of inference in Bayesian networks makes the inference very expensive. Additionally, as we assume a discriminative approach, every possible modification is scored through an evaluation function that must infer the probability distribution of each query variable. Because of those issues, inference time dominates the cost of the revision process.

- Search for literals to be added to a clause: Similarly to FORTE, PFORTE also considers addition of antecedents in clauses in two revision operators, namely add antecedents and add rules. PFORTE uses FORTE's top-down FOIL approach. As in first-order logic theory revision, this approach yields a large number of literals to be added to clauses during the revision of BLPs.

We propose several contributions in order to make the revision of BLPs feasible. First, we noticed that the number of revision points can be reduced. We observe that not all random variables in the Bayesian network of a misclassified example are in fact influencing its probability. We would like to identify the variables relevant to the misclassification, so that only the clauses relative to them are candidates to be

modified. To achieve this goal, we employ the d-separation concept (Geiger et al., 1990) through the use of the Bayes Ball algorithm (Shachter, 1998). Bayes Ball is a linear time algorithm that identifies the set of relevant nodes to a query variable, given evidence on some of others variables. We collect the relevant random variables and mark the clauses producing them as revision points. In this way, we expect that only clauses improving misclassification are liable to be modified during the revision process, greatly reducing the runtime.

Second, the inference runtime may also be improved, and consequently the runtime of the whole revision process, if only the requisite nodes to compute a probability distribution of a variable are taken into account, instead of performing inference in the complete network. Moreover, BLP collects all proofs of each example to compose the Bayesian network. Several ground clauses on the set of proofs may share the same features: besides containing the same predicates, the random variables originating from them may also have the same evidence value or the lack of evidence. Thus, we propose to separate the original large network in small networks, where each one contains only the requisite nodes for computing a probability distribution. The requisite nodes are collected through the Bayes Ball algorithm. Ground clauses with the same features are overlapped so that the number of nodes in the network decreases. Finally, those smaller networks may also contain same information for more than one query variable. Then, we also overlap networks with exactly the same features (same graph and same evidence).

Third, and motivated by the great reduction in runtime resulting by the use of the Bottom Clause in first-order theory revision, we also compose the search space of possible literals to be added to clauses with the literals coming from the Bottom Clause. As BLP does not make a logical difference between positive and negative examples, in the sense the class of the example is the value of the random variable, a Bottom Clause may be built from examples of any class.

We named the revision system built upon those modifications as *BFORTE*, where *B* stands for BLP, Bayes Ball and Bottom clause.

1.3 Publications

The following publications arose from work conducted during the course of this thesis research:

- “Using the Bottom Clause and Mode Declarations in FOL Theory Revision from Examples”, published in Machine Learning Journal (2009) (Duboc et al., 2009) and presented at 18th International Conference on Inductive Logic Programming (ILP-2008) (Duboc et al., 2008). The main ideias and algorithms of this paper are presented in chapter 3.
- “Chess Revision: Acquiring the Rules of Chess through Theory Revision from Examples”, that I presented at 19th International Conference on Inductive Logic Programming (ILP-2009) (Muggleton et al., 2009b). A preliminar version of this paper was also presented at the Workshop on General Game Playing / 21st International Joint Conference on Artificial Intelligence (IJCAI-09) (Muggleton et al., 2009a). A more detailed version of these papers is presented in Chapter 4.
- “ILP through Propositionalization and Stochastic k-term DNF Learning” (Paes et al., 2006b), that I presented at 16th International Conference on Inductive Logic Programming (ILP 2006) and motivated me to start the studies on Stochastic Local Search algorithms. This paper employes stochastic local search over propositionalized versions of the set of examples. The main ideias of this paper are reviewed in the last section of chapter 5.
- “Revising First-order Logic Theories from Examples through Stochastic Local Search” (Paes et al., 2007b), that I presented at 17th Annual International Conference on Inductive Logic Programming (ILP-2007). A preliminar portuguese version of this paper won the best paper prize of the VI *Encontro Nacional de Inteligência Artificial* (Paes et al., 2007a). Chapter 6 is a significant extension of these papers, by using the Bottom Clause to bound the search space and including three additional stochastic components on different steps of the revision process.

- "PFORTE: Revising Probabilistic FOL Theories" (Paes et al., 2006a), that I presented at 2nd International Joint Conference (10th Ibero-American Conference on AI, 18th Brazilian AI Symposium), 2006. This paper extends PFORTE system introduced in (Paes et al., 2005b), by including generalization operators to solve misclassification problem. The algorithm and main ideias discussed there are reviewed in the last section of chapter 7.

In work carried out during the thesis, I also co-authored the following papers:

- "Combining Predicate Invention and Revision of Probabilistic FOL theories" (Revoredo et al., 2006), that I presented at 16th International Conference on Inductive Logic Programming (ILP-2006) and was published in the short paper proceedings of the conference. This paper introduces two predicate invention based operators during the revision process. Due to the expensive cost yielded by these operators, we do not consider them when experimenting the revision system designed in this thesis.

- "On the Relationship between PRISM and CLP(BN)" (Santos Costa and Paes, 2009), presented at International Workshop on Statistical Relational Learning (SIM-2009).

- *"Revisando Redes Bayesianas através da Introdução de Variáveis Não-observadas"* (Revoredo et al., 2009), presented at VI *Encontro Nacional de Inteligência Artificial*(ENIA-2009).

1.4 Thesis outline

The thesis is organized as follows.

Chapter 2 reviews Inductive Logic programming and Theory revision. Basic concepts of ILP and the ideas behind the Bottom clause construction are presented in this chapter. Also, we review key concepts of theory revision from examples and extensively discuss the FORTE system, which is the base system employed in this thesis. We present the top-level algorithm and the algorithms for searching revision points and for proposing modifications to the revision points through revision operators.

Next, in chapter 3 we present the first contribution of this thesis: the introduction of the bottom clause and mode declarations in FORTE and user controlled revision. Part of this chapter was published in (Duboc et al., 2008; Duboc et al., 2009).

Chapter 4 presents the framework we have developed to acquire the rules of variants of chess from the rules of traditional chess, with this problem tackled from the theory revision point of view. We also present additional improvements on the revision system, so that it includes abduction to modify theories and negation in the clauses. This work is published in (Muggleton et al., 2009b; Muggleton et al., 2009a; Muggleton et al., 2009c).

Chapter 5 reviews the main concepts and algorithms of Stochastic Local search. We also present a contribution to propositionalization and stochastic local search to induce definite logic programs, published in (Paes et al., 2006b).

In chapter 6 we present another contribution to first-order theory revision, which is the introduction of stochastic local search in the key searches of the revision process: search for revision points, for the literals to be added/removed from a clause and search for the best revision. Part of this work is published in (Paes et al., 2007a; Paes et al., 2007b).

Chapter 7 reviews main concepts of Bayesian networks, Bayesian Logic Programs and our revision system PFORTE as published in (Paes et al., 2005b; Paes et al., 2006a).

Chapter 8 presents the BFORTE revision system, built upon PFORTE with significant improvements, namely, the use of the Bottom Clause to bound the search space of new literals, the thecniques developes to reduce the inference space and the reduction of the space of clauses to be modified, compared to PFORTE and BLP learning algorithm.

Finally, we conclude the thesis and discuss future work in chapter 9.

The digital version of this thesis can be found in `www.cos.ufrj.br/~ampaes/thesis_ampc.pdf`. Systems developed in this work and datasets used to experiment on them are going to be available in `www.cos.ufrj.br/~ampaes/theory_revision`.

Chapter 2

Inductive Logic Programming and Theory Revision

Machine learning is a sub area of artificial intelligence which studies systems that improve their behavior over time with experience (Mitchell, 1997). Standard machine learning algorithms are propositional, searching for patterns from data of fixed size and represented as attribute-value pairs. In this way, they assume the objects of the domain are homogeneous and sampled from a simple relation. However, real data usually contain many different types of objects and multiple entities, disposed typically in several related tables. *Inductive Logic Programming (ILP)* (Muggleton, 1992; Lavrac and Dzeroski, 1994; Muggleton and De Raedt, 1994; Nienhuys-Cheng and De Wolf, 1997; De Raedt, 2008), also known as Multi-Relational Data Mining (Dzeroski and Lavrac, 2001), combines machine learning and logic programming to automatically induce sets of first-order clauses (*theory*) from multi-relational data. ILP systems have been experimentally tested on a number of important applications (King et al., 1995b; Srinivasan et al., 1997; Muggleton, 1999; Fang et al., 2001; King et al., 2004). Theories learned by ILP elegantly and expressively represent complex situations, even involving a variable number of entities and relationships among them. Such theories are formed with the goal of discriminating between positive and negative examples, given background knowledge. The background knowledge consists of a list of logical facts and/or a set of inference rules.

ILP algorithms assume the background knowledge is fixed and correct. How-

ever, it may be the case that background knowledge also contains incorrect rules and therefore it would be necessary to *modify* them so that they become correct. This is the task of *Theory revision* which has as goal to fix the rules responsible for the misclassification of some example. In this chapter, we start by reviewing the main techniques used in ILP community in section 2.1. Next, we discuss theory revision in section 2.2 with special attention to the system used in this thesis, the FORTE system (Richards and Mooney, 1995). The reader familiar with ILP and Theory Revision may skip this chapter.

2.1 Inductive Logic Programming

Inductive Logic Programming algorithms have as goal to induce from a set of training examples a logic program describing a relational domain, using concepts defined in the *background knowledge*. The returned logic program will be used to classify new examples into positive or negative, using techniques such as resolution. A brief description of some logic programming terms can be found in Table 2.1 and more details can be explored in (Sterling and Shapiro, 1986; Lloyd, 1987; Flach, 1994; Nilsson and Maluszynkski, 2000). The learning problem in ILP is usually defined as follows.

Table 2.1: Some standard Logic Programming terms and their definitions

Term	Definition
constants	Symbols for denoting individuals. Following Prolog convention, we represent constants in lower case.
variables	Symbols referring to an unspecified individual. Following Prolog convention, we represent variables in upper case.
predicate	Symbols for denoting relations, such as mother, loves, etc.
term	They are constants, variables or functions in predicates.
atom	Formulas in the form $p(t_1, ..., t_n)$, where p is a predicate and $t_1, ..., t_n$ are terms.
ground atom	An atom which contains no variable
clause	A formula $\forall(L_1 \vee ... \vee L_n)$, where each L_i is a an atom (positive literal) or the negation of an atom (negative literal)
definite clause	A clause with exactly one positive literal, in the form $A_0 \leftarrow A_1, ..., A_n$, or equivalently $A_0 \vee \neg A_1 \vee ... \vee \neg A_n$, where $n \geq 0$. A_0 is the *head* of the clause, whereas $A_1, ..., A_n$ is the *body* of the clause.
fact	A definite clause where n = 0.
Horn clause	A clause with at most one positive literal.
Herbrand universe	The set of all ground terms constructed from functors and constants in a domain
Herbrand base	The set of all ground atoms over a domain

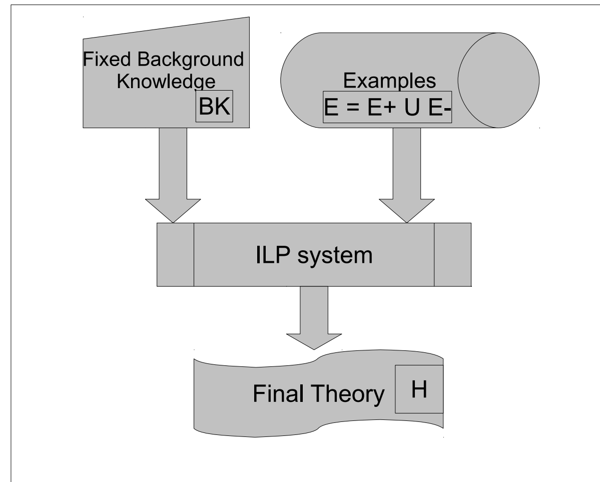

Figure 2.1: Schema of the learning task in ILP, where BK is background knowledge, E is the set of positive (E^+) and negative (E^-) examples and H' is the theory learned by the ILP system.

Definition 2.1 *Given:*

- *A set E of examples, divided into positive E^+ and negative E^- examples and*
- *Background knowledge BK,*

both expressed as first-order Horn clauses.

Learn:

- *A hypothesis H composed of first-order definite clauses such that $BK \wedge H \models E^+$ (H is complete) and $BK \wedge H \not\models E^-$ (H is consistent), i.e., H is correct.*

Figure 2.1 brings a schema of the learning task in ILP.

Often it is not possible to find a correct hypothesis and then the criteria $BK \wedge H \models E^+$ and $B \wedge H \not\models E^-$ are relaxed.

2.1.1 Learning Settings in ILP

The standard learning setting previously defined is known as *learning from entailment* (Frazier and Pitt, 1993). ILP algorithms may also follow the *Learning from interpretations* approach (Valiant, 1984; Angluin et al., 1992; De Raedt and Džeroski, 1994), where the examples are Herbrand interpretations and the goal is to induce a true hypothesis H in the minimal Herbrand model of $BK \wedge E$. In this setting, it is assumed the examples are completely specified. Otherwise, the learning is done from partial interpretations (Fensel et al., 1995) and H must be true in the Herbrand model created by extending $BK \wedge E$. The generalization of learning from partial interpretations is called *learning from satisfiability* (De Raedt, 1997). In this last case, it is required that $H \wedge BK \wedge E \nvDash \Box$.

From the point of view of examples representation, there are two learning settings (De Raedt, 1997):

1. Extensional ILP. An example $e \in E$ is a ground atom. In this case, the whole BK is shared by all the examples, i.e., the conditions are the same for each example. This setting is reduced to learning from entailment.

2. Intentional ILP. An example $e \in E$ is a definite ground clause. Each example may have different information associated to them, i.e., the BK may be divided into a set of definite clauses shared by all examples and sets of definite clauses restricted to each example. This setting is reduced to learning from satisfiability.

Usually, ILP systems such as Progol (Muggleton, 1995), Aleph (Srinivasan, 2001b), FOIL (Quinlan, 1990), TopLog (Muggleton et al., 2008) learn a single target predicate, are extensional and learn from entailment. ILP algorithms may also be designed to learn a set of target concepts, possibly related to each other. CLAUDIEN (De Raedt, 1997) learns multiple predicates from interpretations. TILDE (Blockeel and De Raedt, 1998) has as goal to learn relational decision trees from complete interpretations. Hyper (Bratko, 1999) learns multiple related predicates from entailment. We refer the reader to (Lavrac and Dzeroski, 1994; Dzeroski and Lavrac, 2001; De Raedt, 2008) for more details on ILP systems.

2.1.2 Ordering the Hypothesis Search Space

In order to generalize and/or specialize hypothesis in ILP it is used the framework of θ-subsumption. A clause c_1 *θ-subsumes* a clause c_2 if and only if $\exists$ a variable substitution θ such that $c_1\theta \subseteq c_2$. Due to θ-subsumption, ILP algorithms may traverse the search space from bottom to up (generalizing the hypothesis), from top to down (specializing the hypothesis) or even combining both specialization and generalization strategies. Top-down ILP systems such as (Shapiro, 1983), (Quinlan, 1990) and (De Raedt and Dehaspe, 1997) search the hypothesis space by considering, at each iteration, all valid refinements to a current set of candidate hypotheses. The hypotheses are then evaluated considering their coverage of positive and negative examples, and also measures such as the information gain or the length of the clauses. This strategy considers the set of examples only to evaluate the candidate hypotheses but no to create new hypotheses.

On the other hand, algorithms following a bottom-up strategy use the examples to propose hypotheses. They start with the most specific hypothesis and proceed to generalize it until no further generalizations are possible, without covering some negative examples. Usually, they are based on (1) the least general generalization relative to the background knowledge (RLGG) (Plotkin, 1971), such as it is done in Golem system (Muggleton and Feng, 1990) and its descendant ProGolem (Muggleton et al., 2010) or (2) inverse resolution (Muggleton, 1987; Muggleton and Buntine, 1988; Muggleton and De Raedt, 1994). Nowadays, it is common to somehow combine the bottom-up and top-down strategies, in order to exploit the strengths of both techniques while avoiding their weaknesses. This is the case of systems such as Progol (Muggleton, 1995), Aleph (Srinivasan, 2001b), CHILLIN (Zelle et al., 1994), BETH (Tang et al., 2003) and TopLog (Muggleton et al., 2008), among others.

2.1.3 Mode Directed Inverse Entailment and the Bottom Clause

The bottom clause (Muggleton, 1995) $\perp(e)$ with regard to a clause e and background theory BK is the most specific clause within the hypothesis space that covers the

example e, i.e.,

$$BK \cup \perp (e) \models e \tag{2.1}$$

Any single clause hypothesis covering the example e with regard to BK must be more general than $\perp (e)$. Any clause that is not more general than $\perp (e)$ cannot cover e and can be safely disregarded. Thus, the bottom clause bounds the search for a clause covering the example e, as it captures all relevant information to e and BK. A top-level algorithm of the bottom clause construction process defined in (De Raedt, 2008) is reproduced here as Algorithm 2.1.

Algorithm 2.1 Top-level Algorithm of the Bottom Clause Construction Process (De Raedt, 2008)

1: Find a skolemization substitution θ for e with regard to BK
2: Compute the least Herbrand model M of $BK \cup \neg body(e)\theta$
3: Deskolemize the clause $head(e\theta) \leftarrow M$
4: **return** the result of step 3

In order to understand how the bottom clause is constructed, consider the example e as

$$nice(X) \leftarrow dog(X).$$

and the BK as

$$animal(X) \leftarrow pet(X).$$

$$pet(X) \leftarrow dog(X).$$

First of all, notice that $BK \cup \perp (e) \models e$ is equivalent to $BK \cup \neg e \models \neg \perp (e)$. The first step is to replace all variables in $\neg\, e$ by distinct constants not appearing in the clause (skolemization). The result is one false ground fact coming from the head of $\neg\, e\theta$, since it is a definite clause and a set of positive ground facts coming from the body of $\neg\, e\theta$. In the example above, considering the skolemization substitution as $\theta = \{X \leftarrow skol\}$, we have

$$\neg e\theta = \{\neg nice(skol), dog(skol)\}$$

The next step is to find the set of all ground facts entailed by $BK \cup \neg e$, i.e., the ground literals which are true in all models of $BK \cup \neg e$. This is achieved by com-

puting the least Herbrand model of BK and $\neg body(c\theta)$. In the example, we have

$$\neg \perp (e)\theta = \neg\ e\theta \cup \{pet(skol), animal(skol)\}$$

Finally, each skolemization constant is replaced by a different variable in $\neg \perp (e)\theta$ and the result is negated to obtain $\perp (e)$. In the example,

$$\perp (e) = nice(X) \leftarrow dog(X), pet(X), animal(X)$$

In general, $\perp$ could have an infinite cardinality. Thus, *Mode Directed Inverse Entailment* (Muggleton, 1995) systems such as Aleph and Progol, consider a set of user-defined mode declarations together with other settings to constrain the search for a good hypothesis. A mode declaration (Muggleton, 1995) has either the form $modeh(recall, atom)$ or $modeb(recall, atom)$, where recall is an integer greater than 1 or $'*'$ and atom is a ground atom. $Modeh$ declarations indicate predicates appearing in the head of clauses and *modeb*, predicates in the body of clauses. *Recall* is the maximum number of different instantiations of *atom* allowed to appear in a clause (where '*' means an indefinite number of times). Terms in the atom are either normal or place-marker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type, where type is a constant defining the type of term. The meaning of +, - and # is as follows.

- Input (+) - an input variable of type T in a body literal B_i appears as an output variable of type T in a body literal that appears before B_i, or appears as an input variable of type T in the head of the clause.

- Output($-$) - an output variable of type T in the head of the clause must appear as an output variable of type T in any literal of the body of the clause.

- Constant(#) - an argument denoted by $\#T$ must be ground with terms of type T.

The Algorithm 2.2 illustrates in more details the construction of the bottom clause in Progol (Muggleton, 1995) system, considering modes declaration. The list

$InTerms$ keeps the terms responsible for instantiating the input terms in the head of the clause and the terms instantiating output terms in the body of the clause. The function $hash$ associates a different variable to each term. The depth of a variable v in a definite clause C is 0 if v is in the head of C and $(max_{u \in U_v} d(u)) + 1$ otherwise, where U_v are the variables in the body of C containing v.

Algorithm 2.2 Bottom clause construction Algorithm (Muggleton, 1995)

Input: Background knowledge BK, a positive example e, where $\neg e$ is a clause normal form logic program $\neg a, b_1, ..., b_n$

Output: The bottom clause BC

1: $InTerms \leftarrow \emptyset$, $\bot \leftarrow \emptyset$
2: $i \leftarrow 0$, corresponding to the variables depth
3: $BK \leftarrow BK \cup e$
4: find the first modeh h such that h subsumes e with substitution θ
5: **for each** v/t in θ **do**
6:     **if** v is a $\sharp$ type **then**
7:         replace v in h by t
8:     **if** v is a $+$ or $-$ type **then**
9:         replace v in h by v_k, where v_k is a variable such that $k = hash(t)$
10:     **if** v is a $+$ type **then**
11:         $InTerms \leftarrow InTerms \cup t$
12: $\bot \leftarrow \bot \cup h$
13: **for each** modeb declaration b **do**
14:     **for** all possible substitution θ of arguments corresponding to $+$ type by terms in the set $InTerms$ **do**
15:         **repeat**
16:             **if** b succeeds with substitution θ' **then**
17:                 **for each** v/t in θ and θ' **do**
18:                     **if** v corresponds to $\sharp$ type **then**
19:                         replace v in b by t
20:                     **else**
21:                         replace v in b by v_k, where $k = hash(t)$
22:                     **if** v corresponds to $-$ type **then**
23:                         $InTerms \leftarrow InTerms \cup t$
24:             $\bot \leftarrow \bot \cup \overline{b}$
25:         **until** reaches recall times
26: $i \leftarrow i + 1$
27: Go to line 13 if the maximum depth of variables is not reached
28: **return** $\bot$.

Suppose, for example, the modes declaration below, expressing a fatherhood relationship, where the first argument is the recall number

$modeh(1, father(+person, +person))$

$modeb(10, parent_of(+person, +person))$

$modeb(10, parent_of(-person, +person))$

and the background knowledge

$parent_of(jack, anne)$

$parent_of(juliet, anne)$

$parent_of(jack, james)$

$parent_of(juliet, james)$

and the example $father(jack, anne)$. The most specific clause is

$\bot = father(jack, anne) \leftarrow parent_of(jack, anne), parent_of(juliet, anne)$

where the first literal on the body was obtained by $modeb(10, parent_of(+person, +person))$ and the second literal by $modeb(10, parent_of(-person, +person))$.

The bottom clause with the constants replaced by variables is

$\bot = father(A, B) \leftarrow parent_of(A, B), parent_of(C, B)$

The space complexity of the bottom clause is the cardinality of it and is bounded by $r(|M|j + j-)^{ij^+}$, where $|M|$ is the cardinality of M (the set of modes declarations), j^+ is the number of + type occurrences in each modeb in M plus the number of – type occurrences in each modeh, j^- is the number of – type occurrences in each modeb in M plus the number of + type occurrences in each modeh, r is the recall of each mode $m \in M$, and i is the maximum variable depth (Muggleton, 1995). For more details on formal definitions of the Bottom Clause and Inverse Entailment we refer the reader to (Muggleton, 1995).

2.2 First-order Logic Theory Revision from Examples

ILP algorithms learn first-order clauses given a set of examples and a static and assumed as correct background knowledge. On the other hand, *theory revision from examples* (Wrobel, 1996) has as goal to improve a previously obtained knowledge. To do so, theory revision assumes the provided BK may also contains incorrect rules, which should be modified to better reflect the set of examples. Revision in certain clauses of the BK can be avoided by letting a part of the preliminary knowledge be defined as correct and invariant. Thus, in theory revision the BK is divided in two parts: A set of rules assumed as correct and therefore not modifiable, called here as *Fundamental Domain Theory (FDT)* (Richards and Mooney, 1995); and the remaining rules which may be incorrect and are subject to modifications, called the *Initial Theory*. The goal of a theory revision process is to identify points in the initial theory which prevent it from correctly classifying positive or negative examples, and propose modifications to such points, so that the revised theory together with the FDT is as close to a correct theory as possible. The task of theory revision from examples is defined as follows (Wrobel, 1996).

Definition 2.2 *Given:*

- *A background knowledge BK divided into*
  - *A modifiable set of clauses which might be incorrect (H') and*
  - *An invariant and assumed as correct set of clauses (FDT) and*
- *A set of positive E^+ and negative examples E^- composing the set of examples E*

both written as logic programs;
Find:

- *A revised theory H consisting of definite first-order clauses such that $FDT \wedge H \models E^+$ (H is complete) and $FDT \wedge H' \not\models E^-$ (H is consistent), i.e., H is correct and obeys a minimality criteria*

The minimality criteria may be seen as the attempt of obtaining a revised theory as syntactically and semantically close as possible from the original BK. Usually it is not possible to find a correct theory and such a criteria is relaxed to find a theory as close to be correct as possible.

The schema of theory revision is shown in Figure 2.2. Note that ILP could be seen as a subset of theory revision, where H' is empty and therefore $BK = FDT$.

Theory Revision is particularly powerful and challenging because it must deal with the issues arising from revising multiple clauses (theory) and even multiple predicates (multiple target concepts). Additionally, as the initial theory is a good starting point and the revision process takes advantage of it, the theories returned by revision systems are usually more accurate than theories learned from standard ILP systems using the same dataset. Several papers such as (Shapiro, 1981), (Wogulis and Pazzani, 1993), (Richards and Mooney, 1995), (Buntine, 1991), (Towell and Shavlik, 1994), (Adé et al., 1994), (Wrobel, 1996), (Ramachandran and Mooney, 1998), (Garcez and Zaverucha, 1999), (Esposito et al., 2000) show that propositional and first-order theory revision systems are capable of learning more accurate theories than purely inductive systems and using less examples.

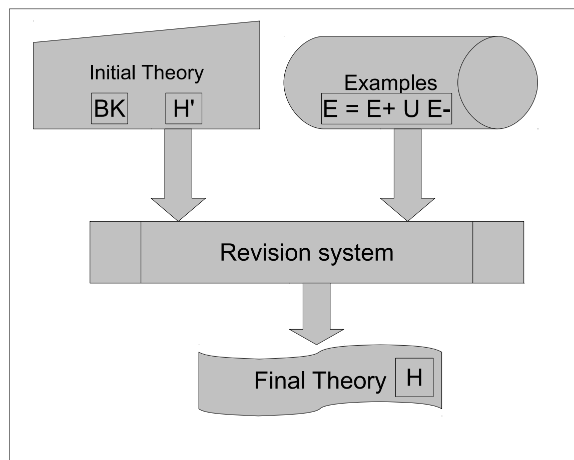

Figure 2.2: Schema of Theory Revision from Examples, where FDT is the fixed preliminary knowledge and H' is the modifiable preliminary knowledge, E is the set of positive (E^{+}) and negative (E^{-}) examples. H is the theory returned by the revision system

.

2.2.1 Revision Points

Usually, the first step in a revision system is to identify the misclassified examples. A positive example not covered by the theory (a *false negative*) indicates the theory is too specific and therefore needs to become more general. In the opposite case, a negative example covered by the theory indicates it is too general and therefore it needs to become more specific. In the former case we need to *generalize* the theory and in the latter case we need to *specialize* the theory.

As the theory may be composed of several target concepts, described by several clauses, it is necessary to find out which clauses and/or literals are responsible for the misclassification of the examples. Also, many clauses can be responsible for proving negative examples as many clauses could be generalized so that the misclassified positive examples become covered. In theory revision, the clauses and literals considered as responsible for misclassifying examples are called *revision points*. It is expected that modifications performed on such points improve the quality of the theory. Depending on the type of the misclassified example being considered we can define two types of revision points:

- *Generalization revision points* - Those are the points in the theory where proofs of positive examples fail.

- *Specialization revision points* - clauses used in successful proof paths of negative examples.

Revision points are the same as *culprit clauses* in MIS system(Shapiro, 1981), which are clauses covering negative examples, found through the SLD tree, and clauses not covering positive examples. In this last case, these clauses are the ones leading to missing clauses, which can be suggested by posing queries to an oracle and using examples it already knows (De Raedt, 2008).

The specification of the revision point determines the type of revision operator that will be applied to make the theory consistent with the dataset. One may consider two types of operators: *generalization* operators, applied on generalization revision points and *specialization* operators, applied on specialization revision points (Wrobel, 1996).

2.2.2 Revision Operators

Theory revision relies on operators that propose modifications at each revision point aiming to transform a theory into another one. Any operator used in first-order machine learning can be used in a theory revision system. In this work we use some operators previously defined in (Richards and Mooney, 1995). Below, we briefly describe them. Next section we show in details how some of theses operators work.

The operators for specialization are:

- *Delete-rule* - this commonly used operator removes a clause that was used to prove a negative example.
- *Add-antecedent* - this operator adds antecedents to an inconsistent clause.

Other approaches exist. Indeed, there are several specialization operators based on the idea of inventing new concepts (predicate invention (Stahl, 1993; Kramer, 1995)) while revising theories (Wrobel, 1994; Bain, 2004).

Different from specialization operators, which only modify existing clauses, generalization operators may create entirely new clauses. As the goal is to cover unprovable positive examples, any ILP operator which accepts positive examples as input can be used. Next we cite usual generalization operators:

- *Delete-antecedent* - this operator removes failed antecedents from clauses that could be used to prove positive examples.
- *Add-rule* - this operator generates new clauses, either from failed existing clauses (deleting antecedents followed by addition of antecedents) or from scratch (starting only from the generalized head of the example).

Other generalization operators exist. One can use *abduction*: first, look for a clause that might satisfy the example but has missing premises, and then add the missing premises. For more details on revision operators we refer the reader to (Richards and Mooney, 1995) and (Wrobel, 1996).

2.2.3 FORTE

In this work we follow the First Order Revision of Theories from Examples (FORTE) system (Richards and Mooney, 1995), which automatically revises function-free first-order Horn clauses. There are several first-order theory revision systems described in the literature, including MIS (Shapiro, 1981), RUTH (Adé et al., 1994), MOBAL (Wrobel, 1993; Wrobel, 1994), INTHELEX (Esposito et al., 2000), among others.

FORTE performs a hill-climbing search through a space of both specialization and generalization operators in an attempt to find a minimal revision to a theory that makes it consistent with the set of training examples. In order to find the revision points FORTE follows a bottom-up strategy, as the training examples are used to find out the clauses/literals presenting some problem. The key ideas of the system are:

1. Identify all the revision points in the current theory using the misclassified training examples.

2. Generate a set of proposed modifications for each revision point using the revision operators. It starts from the revision point with the highest *potential*, defined as the number of misclassified examples that could be turned into correctly classified from a revision in that point. FORTE stops to propose revisions when the potential of the next revision point is less than the score of the best revision to date. Conceptually, each operator develops its revision using the entire training set. However, in practice, this is usually unnecessary and thus FORTE considers only the examples whose provability can be affected after by the revision.

3. Score each revision through the actual increase in theory accuracy it achieves, calculated as the difference between the misclassified examples which turned into correctly classified and the correctly classified examples which turned into misclassified because of the revision.

4. Retain the revision with the highest score.

5. Implement the revision in case the overall score is really improved.

The top-level algorithm is exhibited in 2.3. The algorithm finishes when it cannot find any revision capable of improving the score.

Algorithm 2.3 FORTE Algorithm (Richards and Mooney, 1995)

```
1:  repeat
2:      generate revision points;
3:      sort revision points by potential (high to low);
4:      for each revision point do
5:          generate revisions;
6:          update best revision found;
7:      until potential of next revision point is less than the score of the best
        revision to date
8:      if best revision improves the theory then
9:          implement best revision;
10: until no revision improves the theory;
```

Next we detail the learning setting and representation of the examples in FORTE, followed by the procedures for finding revision points and the algorithms employed by each revision operator.

Examples Representation and Learning Setting

Most ILP algorithms employ the extensional approach to represent examples and background knowledge. In this case, each example is a ground fact and the background knowledge is equally shared by all the examples. FORTE differs from them representing the examples intentionally as they are written as sets of clauses in the format

$$Ground\,Instances\,of\,Target\,Predicates \leftarrow Conjunction\,of\,facts\,from\,the\,context.$$

The head of the above clause is a set of positive and/or negative ground facts, with the same predicate as the concepts intended to be learned. The conjunction of facts from the context is a set of definite clauses confined to the example, i.e., the BK restricted only to that example. There is also a set of background clauses FDT common to all the examples. The FDT and the BK restricted to each example compose the background knowledge of the domain. In this way, each example can be considered as a partial interpretation (De Raedt, 1997).

Although the dataset is composed of partial interpretations, the learning requirement is that the positive instances of each example are logically entailed by the current hypothesis H' together with the background knowledge, i.e., $\forall \mathbf{e} \forall e_+ \in e, H' \wedge FDT \wedge B_e \vDash e_+$ and the negative instances of each example are not logically entailed by the hypothesis and background knowledge, i.e., $\forall \mathbf{e} \forall e_- \in e, H \wedge FDT \wedge B_e \nvDash e_-$, where e_+ represents the positive instances, e_- represents the negative instances and B_e is the background knowledge confined to each example e.

It should be noted from section 2.1.1 that most ILP systems learn a single target predicate only. FORTE, on the contrary, is able to learn multiple predicates simultaneously. Such a task is facilitated because the search space is composed of whole theories instead of individual clauses. When learning individual clauses there is a great chance that the final returned theory is strongly composed of locally optimal and unnecessarily long clauses. On the other hand, when learning whole theories the final hypothesis tends to be globally optimal and smaller. However, learning whole theories is known to be much more expensive than learning individual clauses.

Finding revision points in FORTE

FORTE identifies revision points by annotating proofs of incorrectly provable negative instances or by annotating attempted proofs of incorrectly unprovable positive instances. When the goal is to find the specialization revision points, all the provable instances are considered, since they are the ones whose provability may be affected by a specialization in the theory: any of these instances might become unprovable because of the specialization. These instances are either True Positive (TP) - correctly classified positive instances - or False Positive (FP) - misclassified negative instances. The algorithm for collecting specialization revision points is shown in Algorithm 2.4.

First, FORTE annotates each clause participating in the successful proof of the instances. The positive instances are annotated separately from the negative instances in the clauses. In case the clause has no annotation of false positive instances, it is discarded. The remaining clauses compose the set of specialization

Algorithm 2.4 FORTE Algorithm for collecting specialization revision points (Richards and Mooney, 1995)

Input: The current theory H' and the FDT and EP, the set of provable instances composed of the TP, the set of correctly provable positive instances; FP the set of incorrectly provable negative instances
Output: RPS, a set of clauses marked as specialization revision points, each one annotated with TPC, true positive instances relative to clause C, FPC, false positive instances relative to clause C, and PC the potential of the revision point

```
1: for each provable instance e ∈ EP do
2:     Ce ← clauses participating in the proof of the instance e using H′ and FDT
3:     for each clause C ∈ Ce do
4:         if e ∈ FP then
5:             FPC ← FPC ∪ e;
6:             PC ← PC + 1;
7:         else
8:             TPC ← TPC ∪ e;
9:     RPS ← RPS ∪ Ce;
10: for each clause C ∈ RPS do
11:     if PC = 0 then
12:         delete C from RPS;
```

revision points. The true positive and false positive instances relative to the clauses are used to calculate the score of revision proposed to those points.

In case there are misclassified positive instances, the goal is to find generalization revision points. In this case, all the unprovable instances are considered, since they are the ones whose provability may be affected by a generalization in the theory: any of these instances might become provable because of the generalization. These instances are either True Negative (TN) - correctly classified negative instances - or False Negative (FN) - misclassified positive instances. In order to identify generalization revision points, it is necessary to make annotations from failed proofs of positive instances. Three types of points in the failed proof path are collected:

1. the literal in a clause responsible for the failure proof,
2. the clause whose body contains such literal and
3. the literals which might have contributed to the failure by assigning incorrect values to variables (it is a contribution point).

Thus, each time a backtrack occurs, the failed antecedent is noted and marked as a failure point. Next, the literals binding values to variables in failure points are collected recursively. Finally, the clauses with the failure antecedents and with the contribution antecedents are also marked as failure points. This process is also followed to identify which failure points are responsible for not proving negative instances, since they might become provable after a revision in such points. The list of TN and FN instances are used to calculate the potential of the revision point, and, after proposing some revision, to calculate the score of the revision in such a point. The procedure is exhibited as Algorithm 2.5.

Proposing revisions in FORTE

As stated before, the four basic revision operators add rules, delete rules, add antecedents to clauses and delete antecedents from clauses. FORTE revision operators are ultimately composed of these four basic operations, aggregated to techniques for escaping local maxima, such as deleting/adding several antecedents at once from/to a clause. Additionally, the system uses two operators based on inverse resolution (Muggleton, 1992), namely the absorption and identification operators. The operators are described in terms of the changes they make to the theory. However, recall that each one is *proposing* a revision and not really *implementing* it on the theory. This is only done after proposing all possible revisions and choosing the one with the highest score. In order to calculate the score, FORTE employs a simple evaluation function: the number of incorrect instances which become correct less the number of correct instances which become incorrect because of the revision. Next, we review the way FORTE revision operators work. Note that here we only describe the cases for non-recursive clauses. For more details, including how FORTE deals with recursive clauses, we refer the reader to the original paper (Richards and Mooney, 1995).

Specialization operators

Delete rule The clause marked as a specialization revision point is deleted from the theory. If there is only such a clause explaining a concept, it is replaced by

Algorithm 2.5 FORTE Algorithm for collecting generalization revision points (Richards and Mooney, 1995)

Input: The current theory H' and the FDT; EU, the set of unprovable instances composed of the TN, the set of correctly unprovable negative instances and FN the set of incorrectly unprovable positive instances
Output: RPG, a set of clauses marked as generalization revision points, each one annotated with TNC, true negative instances relative to clause C, FNC, false negative instances relative to clause C, and PC the potential of the revision point

```
1: for each unprovable instance e ∈ EU do
2:     Try to prove instance e using H' and FDT
3:     for each time that a literal fails do
4:         collect the failed literal lfe;
5:         collect the literals LCe responsible for binding variables in lfe, recur-
           sively
6:         collect the clauses Ce where lfe failed and where LCe appeared
7:         if e ∈ TN then
8:             TN_lfe ← TN_lfe ∪ e
9:             for each literal lce ∈ LCe do
10:                TN_lce ← TN_lce ∪ e
11:            for each clause ce ∈ Ce do
12:                TN_ce ← TN_ce ∪ e
13:        else
14:            FN_lfe ← FN_lfe ∪ e
15:            P_lfe ← P_lfe + 1
16:            for each literal lce ∈ LCe do
17:                FN_lce ← FN_lce ∪ e
18:                P_lce ← P_lce + 1
19:            for each clause ce ∈ Ce do
20:                FN_ce ← FN_ce ∪ e
21:                P_ce ← P_ce + 1
22:        RPG ← RPG ∪ lfe ∪ LCe ∪ Ce
23: for each point P ∈ RPG do
24:     if PG = 0 then
25:         delete P from RPG;
```

concept :- fail. This simple process is stated as Algorithm 2.6.

Add Antecedents The operation of adding antecedents to clauses works through two nested processes: the innermost case adds antecedents to the input clause in an attempt to make as many as possible negative instances become unprovable. This specialized clause is included in the revision. In case the specialized clause makes previously provable positive become unprovable, the outer case restarts the special-

Algorithm 2.6 FORTE Delete Rule Revision Operator Algorithm

Input: The current theory H' and the FDT, a clause C marked as specialization revision point together with TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C
Output: A proposed revision *Rev* together with score *Sc*
1: $Rev \leftarrow H' - C$;
2: **if** C is the only clause explaining *concept* **then**
3: $\quad Rev \leftarrow Rev \cup concept : -fail$;
4: $Sc \leftarrow$ calculate_score(Rev, FDT, FPC, TPC)

ization from the original input clause, looking for alternative specializations which retain the proof of positive instances while still making the negative instances unprovable. The top-level process of this revision operator is exhibited as Algorithm 2.7.

Algorithm 2.7 FORTE Top Level Add Antecedents Revision Operator Algorithm

Input: The current theory H' and the FDT, a clause C marked as specialization revision point together with TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C
Output: A proposed revision *Rev*, containing one or more clauses specialized from C, together with score *Sc*
1: $H' \leftarrow H - C$
2: $Rev \leftarrow H'$;
3: **repeat**
4: $\quad C' \leftarrow$ addAntecedents(C, H', FDT, TPC, FPC);
5: $\quad$ **if** C' is different from C **then**
6: $\quad\quad Rev \leftarrow Rev \cup C'$;
7: $\quad\quad FNC \leftarrow$ instances in TPC which become unprovable by $Rev \cup FDT$
8: $\quad\quad TPC \leftarrow FNC$
9: **until** $FNC = \emptyset$ or it is not possible to create a specialized version of C ($C' == C$)
10: $Sc \leftarrow$ calculate_score(Rev, FDT, FPC, TPC)

There are two algorithms for adding antecedents to a clause, which may be executed in replacement to line 4 of Algorithm 2.7:

1. Hill Climbing (Algorithm 2.8) - This algorithm follows FOIL (Quinlan, 1990), adding one antecedent at a time. It works as follows. First, all possible antecedents are created and scored using a slightly modified version of FOIL scoring function, displayed in formula 2.2. There, *Old_score* is the score of the

clause without the literal being evaluated, $\#TPCA$ is the number of positive instances proved by the clause with the literal added to it and $\#FPCA$ is the number of negative instances proved by the clause with the literal. The difference concerns the fact that FOIL score counts the number of proofs of instances, whereas FORTE counts the number of provable instances, ignoring the fact that one instance may be provable in several different ways. Next, the antecedent with the best score is selected. If the best score is better than the current clause score, the antecedent is added to the clause. This process continues until either there are no further antecedents to be added to the clause or no antecedent can improve the current score. This approach is susceptible to local maxima.

$$foil_based_score = \#TPCA * (Old_score - \log(\#TPCA/(\#TPCA + \#FPCA))) \tag{2.2}$$

Algorithm 2.8 Hill climbing add antecedents Algorithm

Input: The current theory H' and the FDT, a clause C, TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C
Output: A (specialized) clause C'
1: **repeat**
2: $antes \leftarrow$ generateAntecedents(C);
3: $Ante \leftarrow$ best antecedent from $antes$, scored with FPC and TPC;
4: **if** $score_(C + \cup\, ante) > score_(C)$ **then**
5: $C \leftarrow C \cup ante$;
6: $FPC \leftarrow FPC -$instances in FPC not proved by C;
7: **until** $FPC = \emptyset$ or it is not possible to improve the score of the current clause
8: **return** C

2. Relational Pathfinding (Algorithm 2.9 - This approach adds a sequence of antecedents to a clause at once in attempt to skip local maxima, as, sometimes, none of the antecedents put individually in the clause improves its performance.

   The Relational Pathfinding algorithm is based on the assumption that generally in relational domains there is a path with a fixed set of relations connecting a set of terms, and such path satisfies the target concept. Its goal is to find

such paths given a relational domain, since important concepts are represented by a small set of fixed paths between terms defining a positive instance. In order to find the paths, the relational domain is represented as a graph where the nodes are the terms and the edges are the relations among them. Thus, a *relational path* is defined as the set of edges (relations) which connect nodes (terms) of the graph. To better visualize such an approach, consider, for instance, the graph in Figure 2.3, which represents part of the family domain. Horizontal lines denote marriage relationships, and the remaining lines denote parental relationships:

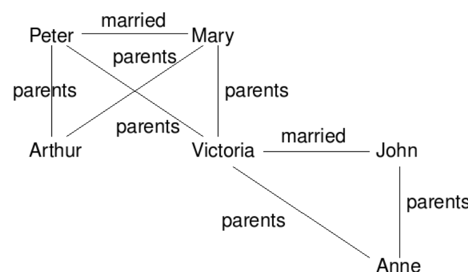

Figure 2.3: An instance of a relational graph representing part of the family domain (Richards and Mooney, 1995)

Now, suppose the goal is to learn the target concept *grandfather*, given an empty initial rule and the positive instance *grandfather(peter,anne)*. The relational path between the terms *peter* and *anne* is composed of the relation *parents* connecting *peter* to *victoria*, and also of the relation *parents* connecting *victoria* to *anne*. From these relations, the path *parents(peter,victoria)*, *parents(victoria,anne)* is formed, which can be used to define the target concept $grandfather(A, B) : -parents(A, C), parents(C, B)$.

From the point of view of theory revision, this algorithm can be used whenever a clause needs to be specialized and it does not have relational paths connecting its variables. In this case, a positive instance proved by the clause is chosen to instantiate it, and, from it, relational paths to the terms without a relationship in the clause are searched.

If the found relations introduce new terms appearing only once, FORTE tries

to complete the clause by adding relations that hold between these singletons terms and other terms in the clause; these new relations are not allowed to eliminate any of the currently provable positive instances. If FORTE is unable to use all of the new singletons, the relational path is rejected.

Algorithm 2.9 Top level Algorithm for Adding Antecedents to a Clause Using Relational Pathfinding Approach

---

Input: The current theory H' and the FDT, a clause C, TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C

Output: A (specialized) clause C'

1: $ex \leftarrow$ a positive instance from TPC, covered only because of C (and not because of others clauses with the same head);
2: find all clauses created from C and from paths generated through the terms in head of ex;
3: $C' \leftarrow$ the clause retaining the most instances in TPC as provable, or, in case of a tie, the shortest clause
4: $FNC \leftarrow$ negative instances still provable
5: **if** $FNC \neq \emptyset$ **then**
6: $\quad C' \leftarrow$ Hill climbing add antecedents algorithm(C', TPC, FPC)
7: **return** C'

Antecedents Generation Following FOIL approach, all the predicates in the knowledge base are considered for creating literals to be added to the clause. A literal is created from a predicate by instantiating its arguments by variables, while respecting the following constraints:

1. At least one variable of the new literal must be in the clause being revised;
2. The arguments of the literals must obey the types defined in the knowledge base.

The larger the number of new variables in the clause is, the more literals are created. Actually, the space complexity grows exponentially in the number of new variables since the complexity of enumerating all possible combinations of variables is exponential in the arity of the predicate.

The algorithms for generating literals used by Hill climbing approach and Relational Pathfinding can be seen in Algorithm 2.10 and Algorithm 2.11, respectively.

Algorithm 2.10 Hill Climbing Antecedents Generation Algorithm

Input: A clause C
Output: A set of literals *antes*
1: **for** each literal *lit* in the knowledge base **do**
2: $varsC \leftarrow$ variables in C with their types;
3: $argsL \leftarrow$ terms in *lit* with their types;
4: **for** each combination *comb* of variables $\in varsC$ in the arity of *lit* **do**
5: **if** *comb* is compatible with $argsL$ considering the types in $argsL$ **then**
6: create a new antecedent *ante* by replacing the terms of *lit* with the variables in *comb*;
7: $antes \leftarrow antes \cup ante$;
8: $n \leftarrow$ arity of *lit* - 1;
9: $i \leftarrow 1$;
10: **while** $i \leq n$ **do**
11: create a new variable v
12: $varsN \leftarrow varsC \cup v$
13: **for** each combination *comb* of variables $\in varsN$ in the arity of *lit*, including at least one variable $\in varsC$ **do**
14: **if** *comb* is compatible with $argsL$ considering the types in $argsL$ **then**
15: create a new antecedent *ante* by replacing the terms of *lit* with the variables in *comb*;
16: $antes \leftarrow antes \cup ante$;
17: $i \leftarrow i + 1$;

As already mentioned, the Relational Pathfinding algorithm starts from a clause grounded from a positive instance covered by the clause. The terms in the ground clause will be the nodes in the graph, connected by the relations defined in the body of the clause. The algorithm constructs the graph iteratively, starting from these initial nodes and expanding them until finding the relational paths. The *end values* are the terms (nodes) created when a node is expanded.

In practice, Relational pathfinding and Hill climbing algorithms might be executed competitively and then the clause chosen in the inner loop is the one with the highest FOIL score. Also, if it is desired, only one of these two approaches may be executed.

Generalization operators

Delete antecedents The algorithm followed by the delete antecedents operator is displayed as Algorithm 2.12.

The antecedents are deleted from a clause marked as a generalization revision point using a hill climbing approach or a method that removes a set of antecedents simultaneously, in case the hill climbing has no success. The generalized clause is added to the proposed revision and the process restarts, in an attempt to make remaining false negative instances become provable, until it is not possible to create useful generalized versions of the original clause. The two methods used to remove antecedents are described below.

1. The first method deletes one antecedent from the clause at each time, following a hill-climbing approach. FORTE chooses the antecedent whose removal makes the largest number of unprovable positive instances become provable, requiring that no unprovable negative instance becomes provable. This process is iterated until there are no antecedents in the clause whose deletion is going to make misclassified positive instances become provable. The process is exhibited as Algorithm 2.13.

Algorithm 2.11 Relational Pathfinding Antecedent Generation Algorithm (Richards and Mooney, 1995)

Input: A clause C
Output: A set of sequence of literals *paths*
1: $CI \leftarrow C$ instantiated with a randomly chosen positive instance;
2: find isolated sub-graphs among the terms in CI;
3: **for** each sub-graph **do**
4:     terms become initial end-values;
5: **repeat**
6:     **for** each sub-graph **do**
7:         expand paths by one relation in all possible ways;
8:         remove paths with previously seen end-values;
9: **until** intersection found or resource bound exceeded
10: **if** one or more intersections found **then**
11:     **for** each intersection **do**
12:         $C' \leftarrow C$ with path-relations added;
13:         **if** C' contains new singleton variables **then**
14:             add relations using the singleton variables;
15:             **if** all singletons cannot be used **then**
16:                 discard C';
17:         replace terms with variables;
18:         $paths \leftarrow paths \cup C'$;

2. Multiple antecedents are removed from the clause at once, in order to overcome the vulnerability of the former approach to get stuck in local maxima. First, the antecedents whose individual removal does not allow true negative instances become provable are collected. Then, combinations of such antecedents are produced, looking for the combination that makes the largest number of positive instances become provable without making negative instances become provable. The process continues deleting antecedents in this way, trying to prove as many positive instances as possible. This algorithm is computationally expensive and it is only executed when the hill climbing approach cannot propose any modification in the current theory. It is shown as Algorithm 2.14.

Algorithm 2.12 Top-level Delete Antecedents Revision Operator Algorithm

Input: The current theory H' and the FDT, a clause C marked as generalization revision point together with TNC, the set of correctly unprovable negative instances because clause C and FNC, the set of incorrectly unprovable positive instances possibly because of clause C

Output: A proposed revision Rev, containing one or more generalized versions of clause C, together with its score Sc

1: $Rev \leftarrow H' - C$
2: $stop \leftarrow false$
3: **repeat**
4: $C' \leftarrow$ hillClimbingDeleteAntecedents(C, TNC, FNC, Rev, FDT);
5: **if** $C' = C$ **then**
6: $C' \leftarrow$ delMultipleAntecedents(C, TNC, FNC, Rev, FDT);
7: **if** $C' = C$ **then**
8: $stop = true$
9: **if** $C' \neq C$ **then**
10: $Rev \leftarrow Rev \cup C'$
11: $FNC \leftarrow FNC-$ instances in FNC which become provable by Rev
12: **if** $FNC = \emptyset$ **then**
13: $stop \leftarrow true$
14: **until** $stop == true$

Add rules FORTE implements two operators for adding rules to the current theory. It may create rules from an existing one or it may create a completely new rule from scratch. In the first case, the operator makes a copy of the clause

Algorithm 2.13 Hill Climbing Delete Antecedents Algorithm

Input: A clause C, TNC, a set of unprovable negative instances, FNC, a set of unprovable positive instances, a theory H'', the static BK FDT
Output: A (generalized) clause C

1: $stop \leftarrow false$
2: **repeat**
3: **for** each antecedent $ante \in C$ **do**
4: $CTemp \leftarrow C - ante$
5: $score_CTemp \leftarrow$ number of instances in FNC which are proved by $H'' \cup CTemp \cup FDT$
6: $score2_CTemp \leftarrow$ number of instances in TNC which are proved by $H'' \cup CTemp \cup FDT$
7: **if** $score_CTemp <= 0$ or $score2_CTemp > 0$ **then**
8: discards $CTemp$
9: $antes \leftarrow (CTemp, score_CTemp)$
10: **if** $antes\ neqemptyset$ **then**
11: $C \leftarrow CTemp \in antes$ with the highest score $score_CTemp$
12: **else**
13: $stop \leftarrow true$
14: **until** $stop = true$
15: **return** C

Algorithm 2.14 Delete Multiple Antecedents Algorithm

Input: A clause C, TNC, a set of unprovable negative instances, FNC, a set of unprovable positive instances, a theory H'', the static BK FDT
Output: A (generalized) clause C'

1: $antes \leftarrow$ all antecedents in C whose deletion does not change TNC
2: **repeat**
3: $ante \leftarrow$ an antecedent in $antes$
4: $CTemp \leftarrow C - ante$
5: **if** no negative instance in TNC become provable by $H'' \cup CTemp \cup FDT$ **then**
6: $C \leftarrow C - ante$;
7: **until** there are no antecedents left to try
8: **if** one or more positive instances in FNC become provable by $H'' \cup CTemp \cup FDT$ **then**
9: **return** C;

marked as a generalization revision point and tries to modify such a copy of the clause in two steps. First, antecedents are deleted from the clause while they make false negative instances become provable. Antecedents are deleted even though true negative instances become provable. Then, the next step of the add rule operator is to add antecedents to the created clause in an attempt to make such negative

instances be unprovable again. This operation is performed by the add antecedents operators as previously explained. The process is exhibited as Algorithm 2.15. The second add rule operator starts by creating the head of the rule from a predicate marked as a generalization revision point. The next step is to compose the body of the clause, using the add antecedents operator.

Algorithm 2.15 Top-level Add rule Revision Operator Algorithm

Input: The current theory H' and the FDT, a clause C marked as a generalization revision point together with TNC, the set of correctly unprovable negative instances because clause C and FNC, the set of incorrectly unprovable positive instances possibly because of clause C

Output: A proposed revision Rev, containing one or more generalized versions of clause C, together with its score Sc

1: $H'' \leftarrow H' \cup C$
2: C' clause C after deleting antecedents which make instances from FNC become provable
3: $H'' \leftarrow H' \cup C'$
4: $TPC \leftarrow$ instances in FNC which become provable by $H'' \cup FDT \cup C'$
5: $FPC \leftarrow$ instances in TNC which become provable by $H'' \cup FDT \cup C'$
6: $Rev' \leftarrow$ addAntecedents(H'', FDT, C', TPC, FPC);

Chapter 3

YAVFORTE: A Revised Version of FORTE, Including Mode Directed Inverse Entailment and the Bottom Clause

3.1 Introduction

Theory revision systems usually induce more accurate theories than ILP techniques learning from scratch. However, such more accurate theories come at the expense of searching in a large search space, mainly because theory revision refines whole theories instead of individual clauses and this is known to be a hard problem (Wrobel, 1996; Bratko, 1999). Therefore, it is essential to develop efficient theory revision systems so that the advantages of them become feasible. Focusing on FORTE theory revision system, in this chapter we contribute towards this goal by identifying a number of the bottlenecks of the revision process and developing algorithms based on state-of-the-art ILP systems to overcome it. The worst bottleneck is related to generation of literals to be included in the body of clauses, which is done inside the add antecedents and add rules operators. FORTE followed the FOIL top-down approach (Quinlan, 1990), considering all the literals of the knowledge base to create antecedents to clauses, which leads to a huge search space, dominating the cost of the revision process. Instead of following a pure top-down approach when specializing clauses, ILP algorithms such as Progol (Muggleton, 1995) and Aleph (Srinivasan, 2001b) restrict the search for literals to those belonging to the *Bottom Clause*. The

Bottom Clause contains the literals relevant to a positive example, collected from a Mode Directed Inverse Entailment (MDIE) search in the BK. This hybrid bottom-up and top-down approach often generates much fewer literals, and they are also guaranteed to cover at least one positive example (the one used to generate the Bottom Clause) and for this reason it is a good option to reduce the search space for literals in the revision process. As part of this work, in (Duboc, 2008) and (Duboc et al., 2009) the Bottom Clause approach is introduced as the search space of literals to speed up FORTE. We describe that process here and also how to further take advantage of the Bottom Clause by (1) allowing its generation to start from a base clause and (2) using the current theory to create literals, in addition to the BK. Moreover, we use mode directed search when deleting antecedents from a clause so that after the deletion the clause is still valid according to the mode declarations.

Besides following FOIL's pure top-down approach, FORTE has six different revision operators and may try to propose revisions suggested by all of them in a single step, depending on the revision points found. This may lead to a large search space of operators. In many tasks, the expert of the domain has some idea about the kind of modifications the theory needs to represent. This could be used as an advantage for reducing the search space of operators. Another weakness of FORTE is that the delete antecedent operator does not allow any incorrectly provable negative example to become unprovable, which may cause overfitting of the clause or not generate a revision by this operator because of such a hard requirement.

In this chapter we describe a number of modifications performed on the FORTE system to handle the shortcomings listed above. We call the resulting system as YAVFORTE (Yet Another Version of FORTE), which includes *FORTE_MBC* (Duboc, 2008; Duboc et al., 2009) to create the search space of new literals. The chapter starts by depicting the modified top level revision process in section 3.2. Next, modifications implemented on the revision operators are devised in section 3.3, concerning mainly the use of Bottom Clause and Mode Directed Search. Experimental results are shown in section 3.4, followed by conclusions about this work in section 3.5.

3.2 Restricting the Search Space of Revision Operators

The original FORTE system proposes modification in the theory through six different revision operators, four designed to generalize the theory and two designed to specialize the theory. Moreover, it tries to apply all these possible operators, even when one of them has already achieved its maximum potential[1] or a maximum score. Suppose for example that we have a clause proving 10 positive examples and 20 negative examples. Now, suppose an extreme case where these 10 positive examples are also covered by another clause. Therefore, the simplest way to rectify the theory would be deleting that clause. However, besides FORTE proposing the deletion of the clause, it would also try to add antecedents to the body of the clause, so that the negative examples become unprovable, which is clearly a waste of time. Thus, in order to make the revision proposals more flexible and efficient, we modified the revision process by including two amendments:

1. The user is able to stipulate which operators the system must apply in order to propose modifications in the theory. In case the user does not specify any operator, the system tries to apply all of them, according to the revision points. Thus, the revision process is able to only specialize/generalize the theory, as well as to apply a subset of specialization and/or generalization operators when proposing modifications.

2. Instead of applying all possible revision operators on a revision point, the system establishes an order of simplicity to apply the operators. Using this order, in case a simpler operator has already achieved the maximum potential or a maximum score, the system stops to propose modifications. Put in other words, the system only applies a more complex operator when no simpler operator simpler than it was able to attain the maximum potential or score. The order imposed to specialization operators is first to apply the delete rule and then the add antecedents operator, clearly because the number of operations

[1] Remember from the previous chapter that the potential is the number of examples indicating the necessity of revision in one point.

performed to evaluate the delete rule is much less than when proposing addition of antecedents. The order imposed to generalization revision operators is delete antecedents, absorption, identification, add rule. Delete antecedents has as search space only the literals belonging to the initial clause. Identification and absorption defines their search space in terms of the literals presented in the theory. Add rule first uses delete antecedents and then add antecedents operators. This last operator may have a large number of literals to evaluate, depending on the background knowledge and mode definitions, therefore we assume it as the more complex generalization revision operator.

Algorithm 3.1 presents the modified revision process, built upon Algorithm 2.3 of chapter 2.1.

Algorithm 3.1 YAVFORTE Top-Level Algorithm

Input: An initial theory T, background knowledge FDT, a set of examples E, list of applicable operators Rev

Output: A revised theory T'

1: **if** $Rev = \emptyset$ **then**
2: $Rev \leftarrow$ all revision operators
3: $GenRev \leftarrow$ ordered list of generalization operators in Rev, starting from the simplest one
4: $SpecRev \leftarrow$ ordered list of specialization operators in Rev, starting from the simplest one
5: **repeat**
6: generate revision points;
7: sort revision points by potential (high to low);
8: **for each** revision point RP **do**
9: **if** RP is a specialization revision point **then**
10: **for** each revision operator $RO \in SpecRev$ **do**
11: apply RO in RP
12: compute $score_{RO}$
13: **else**
14: **for** each revision operator $RO \in GenRev$ **do**
15: apply RO in RP
16: compute $score_{RO}$
17: update best revision found;
18: **until** $score_{RO}$ = maximum score or RO achieved maximum potential of RP
19: **if** best revision improves the theory **then**
20: implement best revision;
21: **until** no revision improves the theory;

3.3 Improvements Performed on the Revision Operators

This section starts by reviewing the theory behind Bottom Clause and MDIE, since we use them to restrict the search space of the add antecedents and the delete antecedents operators. Next, the operator developed in (Duboc et al., 2009) to add antecedents is reviewed but also including improvements performed on it in this work. Then, we describe modifications implemented on delete antecedents operator.

3.3.1 Using the Bottom Clause as the Search Space of Antecedents when Revising a FOL theory

FORTE, following FOIL, generates literals to be added to a clause obeying two conditions: (1) the variables of the literals must follow their types defined in the knowledge base and (2) they must have at least one variable in common with the current clause. While this makes the generation of antecedents simple and fast, it also leads to a large search space composed of all the possible literals of the knowledge base. Such a large search space turns the complexity of the add antecedents operation in a clause very high, contributing to the bottleneck of the revision process. Aiming to reduce such cost, we implemented the following modifications to FORTE system:

1. The variabilized Bottom Clause generated by Algorithm 2.2 became the search space of literals, which reduces the search space and also impose the following constraints:
    - Limits the maximum number of different instantiations of a literal (the recall number);
    - Limits the number of new variables in a clause;
    - Guarantees that at least one positive example is covered (the one which generates the Bottom Clause).
2. The mode declarations are used to further constrains the antecedents, which means they only may be added to the clause if their terms respect the mode defined in the knowledge base.

3. Determination definitions of the form

$$determination(HeadPredicate/Arity, BodyPredicate/Arity)$$

state which predicates can be called in the clauses defining $HeadPredicate$.

The modified version of FORTE considering mode declarations and the Bottom Clause is called *FORTE_MBC*. Recall that when specializing a clause, the goal of the operation is to make false positive examples become unprovable while still covering true positive examples. Thus, the Bottom Clause is created immediately before the search for antecedents begins, by saturating a positive example covered by the clause being specialized (called base clause). The Bottom Clause is going to be composed of the literals relevant to at least such a positive example and it is guaranteed to be a super-set of the base clause. The created Bottom Clause becomes the search space for antecedents, which improves the efficiency of the addition antecedents operation since it usually has many fewer literals than the previous space of the whole knowledge base. It is important to emphasize that the constraints of FOIL continue to be met here, as the arguments of the literals in the Bottom Clause must obey their types and there is a linking variable between the literal being added in the clause and the literals of the current clause.

Algorithm 3.2 shows the process of constructing the Bottom Clause in YAV-FORTE. It differs from Algorithm 2.2 when the base clause has a non-empty body, since in this case it is necessary to take into account the terms of the current clause. Note that in (Duboc et al., 2009) the variables of the Bottom Clause were unified with the variables of the base clause *only after* the Bottom Clause had been constructed. Besides this being an expensive process involving lots of backtracks, sometimes it was not possible to find a correct unification. We noticed two of such situations: cases where the example contains two or more equal terms in the head but the base clause has two different variables in their place (1) and when the base clause has no constant in the head but the first `modeh` has a constant, or vice-verse (2). In these cases the Bottom Clause would follow the unification according to the example, differing from the base clause and making more difficult to find a substitution that would match the Bottom Clause and the base clause. Thus, the first

step in Algorithm 3.2 is to find the ground literals of the base clause considering the example but maintaining the substitution of the variables in the base clause. Then, the terms of the base clause are put in the list of terms to be used by the procedure and the Bottom Clause is initialized with the literals of the base clause. The rest of the process is the same as the original algorithm, except that we do not allow the inclusion of a literal being already in the Bottom Clause. Note that if the clause is being constructed from scratch, it is not necessary to keep an association with the base clause passed as input argument. In this case, we completely follow the original algorithm.

Another difference from the algorithm developed in (Duboc et al., 2009) is the input: there, the Bottom Clause is constructed considering only the BK but here we also allow the current theory to be used to produce literals. This is essential in case we have intermediate clauses in the current theory.

Next, we show how the Bottom Clause is used as the space of new literals.

Using the Bottom Clause in Hill Climbing Add Antecedents Algorithm

The Hill Climbing add antecedents algorithm modified from Algorithm 2.6, to take into account the Bottom Clause as search space, is detailed in Algorithm 3.3. Both algorithms differ in three aspects:

1. Algorithm 3.3 has as first step the construction of the Bottom Clause in line 2, as it is used as search space for antecedents. In order to obtain the Bottom Clause from a positive instance correctly proved by the base clause it is used Algorithm 3.2.

2. Such a Bottom Clause becomes the input for antecedents generation in line 4. This procedure is soon going to be explained in details.

3. As the Bottom Clause created in the beginning is not modified during the execution of this algorithm, it is necessary to remove the antecedent added to the clause from the Bottom Clause, in line 11. From this last difference, it follows that the algorithm also stops when there is no more literal left in the Bottom Clause.

Algorithm 3.2 Bottom clause Construction Algorithm in FORTE-MBC

Input: The current theory H' and the FDT, a clause C, Ex, an instance
Output: The Bottom Clause BC

1: **if** C has an empty body **then**
2: $\quad \bot \leftarrow$ Algorithm 2.2(H', FDT, Ex);
3: **else**
4: $\quad InTerms \leftarrow \emptyset$, $\bot \leftarrow \emptyset$
5: $\quad C_{ground} \leftarrow$ instantiation of the clause C using H', FDT, and Ex, with substitution θ maintaining the variables of C
6: $\quad$ **for** each v/t in θ **do**
7: $\quad\quad InTerms \leftarrow InTerms \cup t$
8: $\quad \bot \leftarrow \bot \cup C$
9: $\quad i \leftarrow 0$, corresponding to the variables depth
10: $\quad BK \leftarrow FDT \cup Ex$
11: $\quad$ **for each** modeb declaration b **do**
12: $\quad\quad$ **for** all possible substitution θ of arguments corresponding to + type by terms in the set $InTerms$ **do**
13: $\quad\quad\quad$ **repeat**
14: $\quad\quad\quad\quad$ **if** b succeeds with substitution θ' **then**
15: $\quad\quad\quad\quad\quad$ **for each** v/t in θ and θ' **do**
16: $\quad\quad\quad\quad\quad\quad$ **if** v corresponds to $\sharp$ type **then**
17: $\quad\quad\quad\quad\quad\quad\quad$ replace v in b by t
18: $\quad\quad\quad\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad\quad\quad\quad$ replace v in b by v_k, where $k = hash(t)$
20: $\quad\quad\quad\quad\quad\quad$ **if** v corresponds to $-$ type **then**
21: $\quad\quad\quad\quad\quad\quad\quad InTerms \leftarrow InTerms \cup t$
22: $\quad\quad\quad\quad$ **if** $b \notin C$ **then**
23: $\quad\quad\quad\quad\quad \bot \leftarrow \bot \cup \bar{b}$
24: $\quad\quad\quad$ **until** reaches recall times
25: $\quad i \leftarrow i + 1$
26: $\quad$ Go to line 14 if the maximum depth of variables is not reached
27: **return** $\bot$.

4. There is a parameter specifying the maximum size a clause is allowed to have.

Using the Bottom Clause in Relational Pathfinding Add Antecedents Algorithm

The Relational Pathfinding algorithm adding more than one antecedent at once in a clause and considering as search space the Bottom Clause is exhibited in Algorithm 3.4.

There are only two major differences between Algorithm 3.4 and Algorithm 2.7: the creation of the Bottom Clause from a positive example covered by the

Algorithm 3.3 Hill Climbing Add Antecedents Algorithm Using the Bottom Clause

Input: The current theory H' and the FDT, a clause C, TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C, CL, maximum size of a clause
Output: A (specialized) clause C'

1: $Ex \leftarrow$ an instance from TPC;
2: $BC \leftarrow$ createBottomClause(Ex, H', FDT, C); %use Algorithm 3.2
3: **repeat**
4: $antes \leftarrow$ getAntecedentsfromBC(C, BC);
5: $Ante \leftarrow$ best antecedent from $antes$, scored with FPC and TPC;
6: **if** $score_(C \cup ante) > score_(C)$ **then**
7: $C \leftarrow C \cup ante$;
8: remove $Ante$ from BC
9: $FPC \leftarrow FPC-$instances in FPC not proved by C;
10: **until** $FPC = \emptyset$ or there are no more antecedents in BC or it is not possible to improve the score of the current clause or $|C| = CL$
11: **return** C

base clause happens before the searching for relational paths (1); consequently, the Bottom Clause is used as search space for paths, together with the same positive example used to generate the Bottom Clause and the base clause (2).

Using the Bottom Clause as Search Space for Antecedents Generation

The original FORTE dynamically generates antecedents at each iteration of the add antecedents procedure, since it is necessary to collect the variables of the current clause to create new literals. FORTE_MBC, on the contrary, generates literals statically, at the beginning of the process by creating the Bottom Clause. However, not every literal in the whole Bottom Clause can be added to a current clause in a specific moment. Suppose, for example, the clause $head(A, B)$. whose body is empty is being specialized. The `modeh` definition to this predicate is $modeh(1, head(+ta, +ta))$, indicating that it can be used in the head of a clause and it is allowed only one instantiation of it ($recall = 1$) since we deal with definite clauses, and their arguments are both of input, whose types are ta. Consider the BK and a positive instance producing the Bottom Clause

$$head(A, B) : -body_1(A, C), body_2(B, C), body_3(C, A).$$

Algorithm 3.4 Top-level Relational Pathfinding Add Antecedents Algorithm Using the Bottom Clause

Input: The current theory H' and the FDT, a clause C, TPC, the set of correctly provable positive instances using clause C and FPC, the set of incorrectly provable negative instances using clause C
Output: A (specialized) clause C'

1: $Ex \leftarrow$ an example from TPC, covered only because of C (and not because of others clauses with the same head);
2: $BC \leftarrow$ createBottomClause(Ex, H', FDT, C); %use Algorithm 3.2;
3: find all clauses created from C and from paths generated through the terms in head of ex, considering BC as search space;
4: $C' \leftarrow$ the clause retaining the most instances in TPC as provable, or, in case of a tie, the shortest clause
5: $FNC \leftarrow$ negative instances still provable
6: **if** $FNC \neq \emptyset$ **then**
7: $\quad C' \leftarrow$ Hill climbing add antecedents algorithm(C', TPC, FPC)
8: **return** C'

based on the following mode declaration:

$$\{modeb(*, body_1(+ta, -ta)), modeb(*, body_2(+ta, -ta)), modeb(*, body_3(+ta, -ta)\}$$

which indicates that the clause can have infinite different instantiations of the predicates $body_i$ ($recall = *$) and the arguments of these predicates have both type ta, where the first one is an input term and the second one is an output term. In case every literal in the Bottom Clause is a candidate to specialize the base clause, the literal $body_3(C, A)$ could be added in the current clause. However, notice the variable C is in the place of an input term and therefore such a variable should have appeared before in the current clause, which is not the case. To consider only the FOIL constraint of having a connection variable between the clause and the literal is not enough, since in this case $body_3(C, A)$ would be valid because of the second variable. Note that such a literal is correctly placed inside the Bottom Clause because the variable C appeared before in $body_2(B, C)$. Because of that ILP systems such as Progol and Aleph take advantage of the order of literals in the Bottom Clause when choosing an antecedent to be added to a clause. We follow a different approach for collecting the eligible literals to be added to a clause, from a Bottom Clause. A literal in the Bottom Clause is a candidate to be included in a current

clause if and only if their input variables have already appeared before in another literal of the current clause. Thus, Line 4 of Algorithm 3.3 is composed of two steps: (1) collect the literals of the current Bottom Clause and (2) validate such candidate antecedents to verify if each one of them is actually allowed to be part of the clause, according to their input variables.

In regard to the Relational Pathfinding Algorithm we follow a slightly different approach. First of all, it is important to notice that the generation of antecedents to this algorithm is the same as the one performed in Algorithm 2.9, with only one obvious difference: now the literals are searched in the Bottom Clause generated at the beginning of the process. Thus, when looking for paths the literals considered to be in a path are the ones from the Bottom Clause, but still taking into account the end values of the relational paths. However, we do not validate literals concerning modes immediately before they are considered to be in a path, since more than one antecedent will be added at once. In this way, it may be the case the whole path is valid according to the modes but the isolated literal would not be. Thus, the whole path is validated according to mode declarations: if the relational path does not obey the modes it is discarded just before it is evaluated. Line 3 of Algorithm 3.4 returns the clauses created from C and from paths, but ensuring such clauses are valid according to mode declarations.

Remarks about the Complexity of Antecedents Addition

The revision process of FORTE has an exponential complexity in the size of the input theory and in the arity of the theory predicates (Richards and Mooney, 1995). When adding antecedents to a rule, the number of permutations of arguments to a predicate is an exponential function of the predicate's arity. Thus, the space complexity of possible literals grows exponentially on the number of new variables, since the complexity of enumerating all possible combinations of variables is exponential according to the arity of the predicate. On the other hand, the space complexity of the Bottom Clause is the cardinality of it and is bounded by $r(|M|j + j-)^{ij^+}$, where $|M|$ is the cardinality of M (the set of mode declarations), j^+ is the number of + type occurrences in each modeb in M plus the number of – type occurrences

in each `modeh`, j^- is the number of – type occurrences in each modeb in M plus the number of + type occurrences in each `modeh`, r is the recall of each mode $m \in M$, and i is the maximum variable depth (Muggleton, 1995). On the same way the use of the Bottom Clause brings the advantage of reducing the search space of Progol when compared to a top-down approach as FOIL, the use of the Bottom Clause also reduces the search space of antecedents of FORTE, which originally generated antecedents based on FOIL. However, it is important to point out that this advantage is only guaranteed if the i variable is small enough. In the extreme case that i and the background knowledge are both large, the Bottom Clause has a very large number of literals and hence the search space is as large as or even larger than the search space of FORTE. Considering exactly this problem, (Tang et al., 2003) proposed the BETH system, which makes use of a hybrid top-down and bottom-up approach when constructing the Bottom Clause, aiming to reduce the cardinality of the Bottom Clause and consequently the search space of literals.

3.3.2 Modifying the Delete Antecedent Operator to use Modes Language and to Allow Noise

Using Mode Directed Search when Deleting Literals from a Clause

Besides using the theory of MDIE for generating the Bottom Clause and take advantage of it when adding literals to be added to a clause, we would also need to yield a theory that follows the modes language. This is not only a requirement to make the final theory valid according to the modes, but is also essential to the specialization procedure since, as it was said before, the first step is to include the terms in the base clause as terms to be used in the Bottom Clause. Such terms must correctly follow the modes, otherwise it will not exist a match for them in the Bottom Clause construction procedure.

To begin with, we assume the initial theory follows the modes established in the language bias. To continue following modes when revising the theory, it is necessary to validate each single proposed modification according to them. When specializing clauses, this is already done, since any literal to be added to the body of a clause comes from the Bottom Clause and immediately before to be indeed included, it

is checked if it obeys any mode. It remains to be checked whether a proposed modification does not make the theory invalid according to the modes when it is generalized. Thus, a mode check is included after deleting a literal from the body of the clause being generalized. It works as follows. Immediately before evaluating a literal to be removed from a clause, it is checked if the resulting clause follows one of the modes assertions. The inspection will prevent a literal to be removed in case it falls in one of two cases: (1) the literal to be removed is the only one with an output variable of the head of the clause; or (2) the literal to be removed is the only one with a variable that is input to other literal. Both cases would make the clause not follow the modes language and therefore the procedure does not allow this to happen.

Allowing noise in Delete Antecedent Operator

In addition to using modes definitions to validate a deletion of a literal from a clause, we also remodel the delete antecedents operator so that it becomes more flexible in the sense of allowing *noise*, i.e., true negative instances become provable. Rather than specifying a maximum number of negative examples to be considered as noise, as it is done in Aleph and Progol systems for example, the number of negative examples that a clause may cover is decided by the score of the clause. Thus, at each iteration the antecedent improving the score at most is selected. In case after deleting such an antecedent from the body of a clause the score is improved, yet a true negative instance becomes false positive, the procedure nevertheless proceeds to further deletions. The deletion of antecedents stops when it is not possible to improve the score, following a classical greedy hill climbing approach. This modus operandi is used in deletion antecedents operator and to delete antecedents in order to create a new rule from an existing one (add rules operator).

3.4 Experimental Results

We have already experimentally demonstrated in (Duboc et al., 2009) the benefits of using mode declarations and the Bottom Clause when revising FOL theories. Experimental results have presented a speed-up of 50 times compared to the original

Algorithm 3.5 Remodeled Greedy Hill Climbing Delete Antecedents Algorithm

Input: A clause C, TNC, a set of unprovable negative instances, FNC, a set of unprovable positive instances, a theory H'', the static BK FDT
Output: A (generalized) clause C
1: $score_C \leftarrow$ compute current score
2: **repeat**
3: **for** each antecedent $ante \in C$ **do**
4: $CTemp \leftarrow C - ante$
5: $score_{CTemp} \leftarrow$ compute score for $CTemp$
6: $antes \leftarrow (CTemp, score_{CTemp})$
7: $C \leftarrow CTemp \in antes$ with the highest score $score_{CTemp}$
8: **if** $score_{CTemp}$ is better than $score_C$ **then**
9: $C \leftarrow CTemp$
10: $score_C \leftarrow score_{CTemp}$
11: **until** it not possible to improve score
12: **return** C

FORTE system, without significantly decreasing accuracies. Additionally, we have shown there that the revision system provides more accurate and smaller theories, compared to a standard inductive method also based on the Bottom Clause. In this chapter, we would like to know if it is possible to further decrease the runtime of the revision process without decreasing the accuracy. With this goal, we present the results obtained from the current implementation, varying the set of operators used to revise the theory, compared to the implementation of FORTE containing FORTE_MBC algorithm and also to Aleph system. We compare the average runtime, accuracy and size of the theories, in number of clauses and literals.

Datasets We consider the same datasets used in (Duboc et al., 2009), namely the Alzheimer (King et al., 1995a) domain, composed of four datasets and the DssTox dataset (Fang et al., 2001). Alzheimer domain compares 37 analogues of Tacrine, a drug combating Alzheimer's disease, according to four properties as described below, where each property originates a different dataset:

1. inhibit **amine** re-uptake, composed of 343 positive examples and 343 negative examples

2. low **toxic**ity, with 443 positive examples and 443 negative examples

3. high acetyl **choline**sterase inhibition, composed of 663 positive examples and 663 negative examples and

4. good reversal of **scopolamine**-induced memory deficiency, containing 321 positive examples and 321 negative examples

Alzheimer domain considers 33 different predicates and 737 facts in background knowledge. DssTox dataset was extracted from the EPA's DSSTox NCTRER Database. It contains structural information about a diverse set of 232 natural, synthetic and environmental estrogens and classifications with regard to their binding activity for the estrogen receptor. The dataset is composed of 131 positive examples and 101 negative examples. There are 25 different predicates and 16177 facts in backgrund knowledge.

Experimental Methodology The datasets were split up into 10 disjoint folds sets to use a K-fold stratified cross validation approach. Each fold keeps the rate of original distribution of positive and negative examples (Kohavi, 1995). The significance test used was corrected paired t-test (Nadeau and Bengio, 2003), with $p < 0.05$. As stated by (Nadeau and Bengio, 2003), corrected t-test takes into account the variability due to the choice of training set and not only that due to the test examples, which could lead to gross underestimation of the variance of the cross validation estimator and to the wrong conclusion that the new algorithm is significantly better when it is not. In this work, we are assuming that both modes and types definitions are correct and therefore cannot be modified. All the experiments were run on Yap Prolog (Santos Costa, 2008).

The initial theories were obtained from Aleph system using three settings:

- The first setting runs Aleph with its default parameters, except for `minpos`[2], which was set to 2 to prevent Aleph from adding to the theory ground unit clauses corresponding to positive examples. It is identified in the Tables as **Theory-def**.

[2]Minpos parameter set a lower bound on the number of positive examples to be covered by an acceptable clause. If the best clause covers positive examples below this number, then it is not added to the current theory (Srinivasan, 2001b)

- The second setting named as **Theory-def+cl**, runs Aleph with its default parameters, except for minpos and `clauselength` parameters, to set an upper bound on the number of literals in a clause. We choose this last parameter because YAVFORTE implementation also limits the number of literals in an acceptable clause. To all systems, clause length is defined as 5 in Alzheimer domain and 10 in Dsstox, following previous work (Landwehr et al., 2007).

- The third setting, **Theory-best** runs Aleph with "literature" parameters according to previous work (Huynh and Mooney, 2008; Landwehr et al., 2007). Thus, following (Huynh and Mooney, 2008), Aleph was set to consider `minscore` as 0.6, evaluation function is M-estimate (Dzeroski and Bratko, 1992), noise is 300 for Alzheimers and 10 for DSStox and clause length and minpos are defined as above. Additionally, `induce_cover` command was invoked instead of the standard "induce". The difference is that positive examples covered by a clause are not removed prior to seeding on a new example when using induce_cover. Note that only clause length parameter is used in YAVFORTE, as the others parameters are not implemented there.

To generate such theories, the whole dataset was considered but using a 10-fold cross validation procedure. Thus, a different theory was generated for each fold and each one of these theories was revised considering its respective fold (the same fold is used to generate and revise the theories).

In order to identify if there are any benefit on pre-defining the set of applicable operators when revising the theories, YAVFORTE is run with 5 different sets of operators. Four settings consider a combination of one specialization operator together with one generalization operator. The last setting, identified in tables as $YAVFORTE$ considers all operators.

- **YAV-del** considers delete rule as specialization operator and delete antecedents only as generalization operator.

- **YAV-add** considers only addition of antecedents as specialization operator and addition of rules as generalization operator.

- **YAV-add-del** considers additions of antecedents as specialization operator and delete antecedents as generalization operator.

- **YAV-del-add** considers delete rules as specialization operator and add antecedents as generalization operator.

Both YAVFORTE and FORTE_MBC run add antecedents algorithm considering Relational Pathfinding algorithm, followed by Hill climbing in Alzheimers domains. As DSSTox top-level predicate is unary, Relational Pathfinding is not applicable. Because of that, only Hill climbing algorithm is taken into account. Both systems considers clause length parameter as 5 for Alzheimers and clause length as 10 for DssTox.

Table 3.1: Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer amine dataset

System	Theory-def			Theory-def+cl			Theory-best		
	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses
Aleph	7.53	62.67	20.8 **5.5**	48.66	65.62	36.9 **8.5**	43.19	73.64	69.7 **7.16**
FORTE_ MBC	21.96	71.38	34.6 8.7	26.29	75.09	35.5 8.6	69.09	76.87	52.5 12
YAV-del	**3.91** ♢ • ⋆	70.13 ♢⋆	**18.4** **5.5**	**6.01** ♢ • ⋆	75.09 ♢	**34.7** **8.5**	**12.89** ♢ • ⋆	76.87 ♢	**51.9** 11.8
YAV-add	7.29 •⋆	70.48 ♢⋆	46.7 10.8	8.61 ♢•	74.94 ♢	53.8 11.9	15.31 ♢ • ⋆	76.55 ♢	79 17.2
YAV-add-del	9.14 ♢ • ⋆	71.92 ♢⋆	42.2 9.8	7.13 ♢ • ⋆	75.54 ♢	39.1 9.2	15.93 ♢ • ⋆	76.55 ♢	74.9 16.4
YAV-del-add	7.18 •⋆	72.34 ♢⋆	43.2 10.1	8.93 ♢•	75.24 ♢	52.8 11.7	14.14 ♢ • ⋆	**77.57** ♢	55.5 12.4
YAV-FORTE	12.76 ♢•	**74.23** ♢•	34.6 8.7	11.15 ♢•	**75.68** ♢	38 9	29.12 ♢•	77.27 ♢	58.9 12.8

Results and remarks about them Tables 3.1, 3.2, 3.3, 3.4 and 3.5 bring the results for Amine, Toxic, Choline, Scopolamine and DssTox datasets, respectively. The best values for each column are in bold, but we require that, in case the best runtime or theory size result is of one of the revision settings, the accuracy of the initial theory is still improved. The symbol ♢ identifies the cases where the runtime and accuracy of the new systems have significant difference compared to Aleph. The symbol • is used for a similar reason, comparing YAVFORTE with its

Table 3.2: Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer toxic dataset

System	Theory-def			Theory-def+cl			Theory-best		
	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses
Aleph	17.29	62.48	13 **4.3**	53.57	63.85	32.2 **8.1**	25.98	77.75	40.1 10.4
FORTE_MBC	22.54	**71.32**	25.4 7.8	37.67	71.19	**27.9** **8.1**	22.14	79	33.5 8.9
YAV-del	**2.93** ♢ • ⋆	68.48 ♢	**11.6** 4.5	**8.79** ♢ • ⋆	71.19 ♢	28.1 **8.1**	**4.38** ♢ • ⋆	**79.12** ♢	**32.4** **8.7**
YAV-add	4.21 ♢•	63.62 •⋆	22.3 6.3	16.57 ♢•	75.71 ♢ • ⋆	62.1 14.3	5.92 ♢ • ⋆	77.53 •	51.5 11.8
YAV-add-del	4.64 ♢•	68.94 ♢	14.8 5.1	10.99 ♢ • ⋆	71.64	30.9 8.6	6.13 ♢ • ⋆	77.76 •	51.4 11.8
YAV-del-add	4.69 ♢•	64.30 ♢ • ⋆	20.9 6	22.39 ♢ • ⋆	**75.93** ♢	53.6 12.6	4.42 ♢ • ⋆	78.89 ♢	32.5 **8.7**
YAV-FORTE	6.10 ♢•	69.05 ♢	14.8 5.1	16.73 ♢•	71.64 ♢	30.9 8.6	11.51 ♢•	78.67	44.9 10.6

Table 3.3: Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer choline dataset

System	Theory-def			Theory-def+cl			Theory-best		
	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses
Aleph	39.23	56.02	30.7 **8.5**	100.63	56.62	39.33 **9.4**	147.94	64.47	74.9 17.1
FORTE_MBC	100.73	61.01	34.6 9.8	79.21	64.93	40 10.3	228.18	**67.6**	**15.9** **8.9**
YAV-del	**19.83** ♢ • ⋆	65.21 ♢•	**27.7** 8.9	**24.13** ♢ • ⋆	64.48	**38.7** 10.1	**38.33** ♢ • ⋆	65.09 •	66.2 15.6
YAV-add	37.57 •⋆	64.32 ♢•	66.1 6.3	36.23 ♢ • ⋆	65.16 ♢	67.6 14.3	46.85 ♢ • ⋆	64.70 •	91.1 19.8
YAV-add-del	46.05 •⋆	62.98	60.2 15.2	39.07 ♢ • ⋆	**67.09**	51.9 12.1	59.22 ♢ • ⋆	64.94 •	82.4 18.2
YAV-del-add	39.58 •⋆	**66.32** ♢ • ⋆	63.7 15.2	38.63 ♢ • ⋆	65.39 ♢	65.6 14.4	51.42 ♢ • ⋆	64.93 •	75 16.7
YAV-FORTE	64.36 ♢•	64.02 ♢•	51.7 13.5	70.57 ♢	64.93 ♢	40 10.3	98.76 ♢•	64.86 •	77.8 16.7

difference settings to FORTE_MBC. Finally, ⋆ identifies the cases where there is significant difference between YAVFORTE disregarding some revision operator and YAVFORTE using all the operators. From the results we make remarks as follows.

- YAVFORTE is always faster than FORTE_MBC and even produces more accurate theories with significant difference in four cases. There are two cases where YAVFORTE returns worse theories than FORTE_MBC: Choline and DSSTox with the literature Aleph parameters. In the rest of the cases both

Table 3.4: Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for Alzheimer Scopolamine dataset

System	Theory-def			Theory-def+cl			Theory-best		
	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses
Aleph	15.96	51.71	24.9 **6.5**	41.40	51.41	31.3 **7.5**	57.24	58.42	45.3 10.5
FORTE_MBC	47.43	61.53	37.8 9.7	42.56	**66.53**	37 9.2	48.04	**64.51**	45.1 10.9
YAV-del	**4.97** ♢ • ⋆	60.74 ♢⋆	**22.8** **6.5**	**6.45** ♢ • ⋆	64.79 ♢	**28.3** **7.5**	**7.27** ♢ • ⋆	64.20 ♢	**40.9** **10**
YAV-add	3.54 ♢ • ⋆	51.87 •⋆	28.2 7.4	14.67 ♢•	62.02 ♢ • ⋆	57.5 13	15.02 ♢ • ⋆	62.17 ♢ • ⋆	66.9 14.8
YAV-add-del	5.30 ♢ • ⋆	60.59 ♢⋆	23.70 6.7	7.70 ♢ • ⋆	63.87 ♢ • ⋆	29.8 7.8	8.9 ♢ • ⋆	64.35 ♢	45.8 11
YAV-del-add	3.48 ♢ • ⋆	51.87 •⋆	28.2 7.4	14.8 ♢•	62.02 ♢ • ⋆	57 12.9	14.06 ♢ • ⋆	61.70 ♢ • ⋆	62.6 13.9
YAV-FORTE	13.93 ♢•	**62.47** ♢	37.5 9.6	15.68 ♢•	**66.53** ♢	37 10.3	48.04 ♢	**64.51** ♢	45.1 11.3

Table 3.5: Runtime in seconds, predictive accuracy and size in number of literals and clauses of final theories for DssTox dataset

System	Theory-def			Theory-def+cl			Theory-best		
	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses	Runtime (s)	Acc (%)	#Lits, #Clauses
Aleph	25.55	51.36	**10.2** **2.9**	50.07	51.36	**11.2** **2.9**	39.96	55.39	**9** **2.3**
FORTE_MBC	4.26	71.37	31.4 6	4.27	71.37	31.4 4.2	3.12	**77.43**	20.8 4.2
YAV-del	0.25 ♢ • ⋆	49.08 •⋆	10.2 2.9	0.24 ♢ • ⋆	49.08 •⋆	10.2 2.9	0.15 •⋆	54.73 •⋆	8.4 2.3
YAV-add	2.47 ♢•	78.33 ♢•	40.5 7.4	2.46 ♢•	78.33 ♢•	40.5 7.4	1.5 ♢•	71.34 ♢•	26.2 5
YAV-add-del	2.39 ♢•	75.29 ♢ • ⋆	35.9 6.4	2.38 ♢•	75.29 ♢ • ⋆	35.9 6.4	1.66 ♢•	72.64 ♢•	27.6 5
YAV-del-add	**2.06** ♢•	**79.20** ♢•	33.9 6.6	**2.06** ♢•	**79.20** ♢•	33.9 6.6	**0.57** ♢ • ⋆	61.34 ♢ • ⋆	14.7 3.3
YAV-FORTE	2.56 ♢•	78.33 ♢•	38.9 7.2	2.61 ♢•	78.33 ♢•	38.9 7.2	1.72 ♢•	71.34 ♢•	24 4.7

systems produces similar accuracy results. The difference in time is mainly due to the fact that YAVFFORTE may stop to propose revisions without using all operators, when a simpler operator already reaches the potential of the revision point. Additionally, there is some gain in time when terms of the base clause are included in the set of terms of the Bottom Clause before generating literals. Remember that FORTE_MBC is used to unify the terms of the base clause and the Bottom Clause only after the Bottom Clause has been constructed.

Concerning the difference in accuracy, there are two factors responsible for that: (1) to stop proposing revisions before trying all operators may cause the revision let aside some revision that could be better to the test set. This is the case with Choline dataset; also, delete antecedents operator in YAVFORTE does not try to eliminate the proof of all negative examples; instead it deletes antecedents following a score value. While this more flexible operator is able of producing more accurate results in the four cases previously mentioned, it is also responsible for worse revision of the DSSTox theory. However, in most cases, YAVFORTE is able to revise theories as well as FORTE_MBC (without significant difference) in reduced runtime.

- Note that the results for Choline dataset using FORTE_MBC are in most of the figures detached from the rest. FORTE_MBC has a harder time to revise the theories generated from this dataset than in the others datasets for two main reasons: this is the dataset with the largest set of examples and the theories generated from Aleph for it have the highest size.
- YAVFORTE considering only delete antecedents and delete rules as revision operators achieves the fastest revision system and produces the smallest theories according to both number of literals and number of clauses. What is interesting about this setting is that with rare exceptions (Choline with best parameters of Aleph and DssTox), the results show that those both operators alone are able to provide significant improvements over the initial theory. Actually, in Toxic with literature Aleph parameters this setting is the one providing the best revision: the final theory is the smallest and more accurate and it even has been revised in less time.
- The results indicate that there are benefits of considering only a subset of the revision operators: most cases produce theories as accurate as when considering all operators and in less time. Thus, when the expert of the application has some insight on what to expect from the revised theory, he could use this knowledge to reduce the set of applicable revision operators.
- It is important to emphasize that YAVFORTE considering any set of operators

is faster than Aleph in 11 of 15 cases, with most of the settings still returning more accurate theories. With this result, we can claim that a revision system is capable to behave better than an inductive system, considering both runtime and accuracy.

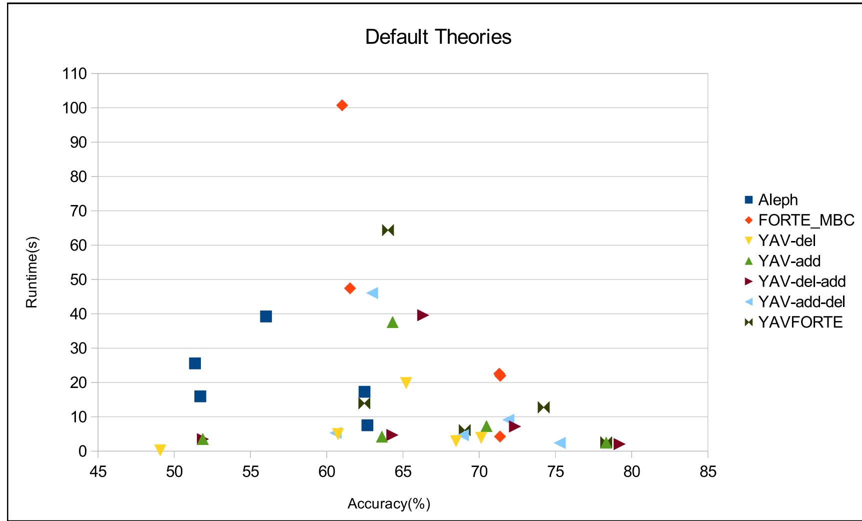

Figure 3.1: Scatter plot for all datasets and systems settings considering theories learned by Aleph with its default parameters

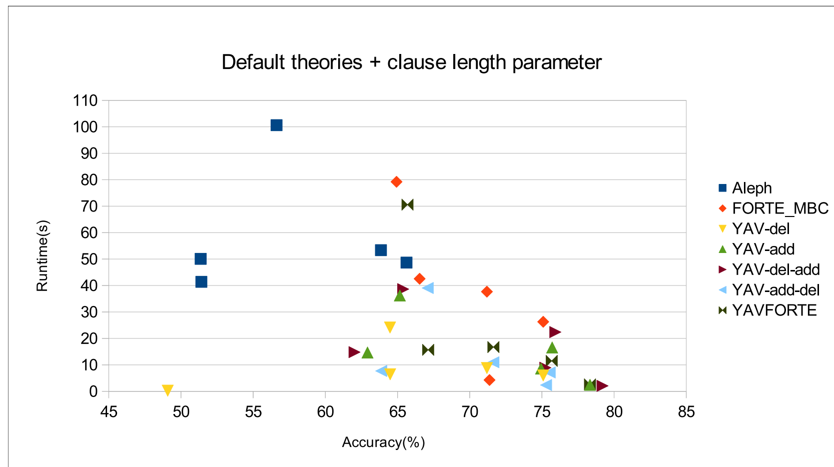

Figure 3.2: Scatter plot for all datasets and systems settings considering theories learned by Aleph with its default parameters, except for clause length

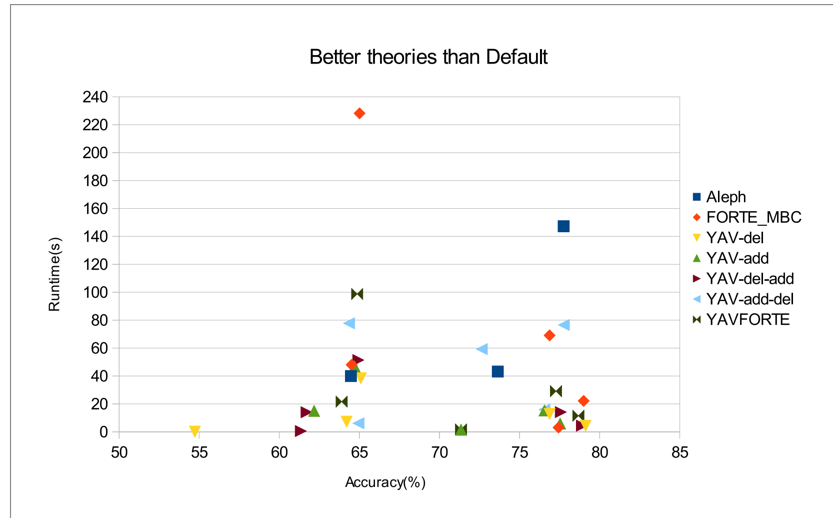

Figure 3.3: Scatter plot for all datasets and systems settings considering theories learned by Aleph with better parameters settings than just default

Figures 3.1, 3.2 and 3.3 exhibit scatter plots for all systems and theories, considering theories learned with Aleph default parameters, Aleph default plus clause length changed and Aleph best parameters, respectively. Symbols in the bottom right corner indicate the best results. Note that when considering default parameters Aleph concentration is in bottom left corner: runtime is small, at the cost of bad accurate theories. When changing clause length parameter, the situation slightly changes for accuracy, but runtime increases, as expected. Better parameters improve accuracy, at the cost of increasing runtime. Most of the cases FORTE_MBC is in higher y-axis than the others systems, although it is more present in the left side, indicating its accuracies results are still elevated, compared to Aleph. YAVFORTE settings, on the other hand concentrates in the bottom (right) corner, indicating in the overall they are capable of producing the best accurate theories, in less time.

3.5 Conclusions

Although work in theory revision gained great attention in the 1990s, in recent years ILP community had practically set aside research in this area, since the benefits of revising theories could not outweigh the large runtime effort expended by those

systems. This chapter contributes towards changing this scenario, based on the FORTE revision system. First, in (Duboc et al., 2009) we have abandoned the top-down search of literals based on FOIL in order to use the Bottom clause, to reduce the set of literals taken into account when refining a theory, as standard ILP systems do. In this chapter we further improve the scalability of the revision system, by (1) making the use of revision operators more flexible, once it is possible to choose which operators are going to be considered to revise the theory; (2) stopping to propose revisions in one revision point as soon as a simpler operator already achieves the full potential of the point; (3) requiring the clause continues to obey mode declaration after an antecedent is deleted.

We additionally introduced one modification in the delete antecedents operator, once in the original FORTE and FORTE_MBC an antecedent could be deleted only when none of the true negative instances become provable. It was necessary to make this hard requirement more flexible because there are cases when the set of false negative instances is only reduced if the clause is allowed to also prove some negative instances. Finally, we have modified the Bottom Clause construction procedure of FORTE_MBC in two ways: (1) the terms of the ground base clause are considered to be part of the Bottom Clause, so that they can be used to bring further terms to the Bottom Clause, and also to make unnecessary to unify the base clause with the Bottom Clause after this last one is constructed; (2) the current theory is taken into account to prove literals to be included in the Bottom clause. We named the system including all these issues as YAVFORTE.

Experimental results were extracted from five relational benchmark datasets, namely four datasets from Alzheimer domain and the DSSTox dataset. Through them, it was possible to verify that the revision process can be indeed improved with those modifications. YAVFORTE is faster than FORTE_MBC without decreasing the accuracy. In fact, there were cases where the accuracy was further improved, considering the initial theory. It was also possible to see that when we turn the set of revision operators more flexible, the runtime diminishes while the initial accuracy is still improved, though not that much as when the complete set of operators are considered. More important, we could show that when the same parameter for

limiting the size of a clause is used for revising and for learning from scratch, the revision performs faster than the inductive method. In this way, we achieve our goal of devising a revision system as feasible as a standard inductive system, at least when datasets of regular size are used.

Datasets used in this chapter are not considered as robust ones. Therefore, we still need to verify how the revision system behaves when the datasets have a large number of examples and/or large background knowledge. Also, as the revision system starts from an initial theory, if this one has a large number of faulty clauses, the system is probably going to behave badly. In this case, it is going to be more difficult to do the revision than learning from scratch, since the revision must propose modifications to each faulty clause. Because of those issues, in chapter 6 we investigate stochastic local search techniques applied to the revision process.

Chapter 4

Chess Revision: Acquiring the Rules of Variants of Chess through First-order Theory Revision from Examples

4.1 Motivation

In the recent paper (Dietterich et al., 2008), the authors point out that there was considerable effort in the development of theory revision systems in the past, but the lack of applications suited to that task made those systemsnot be widely deployed. In this chapter we intend to contribute in this direction by designing an application in the area of games that fits perfectly to theory revision.

Game playing is a fundamental human activity, and has been a major topic of interest in AI communities since the very beginning of the area. Games quite often follow well defined rituals or rules on well defined domains, hence simplifying the task of representing them as computer programs. On the other hand, good performance in games often requires a significant amount of reasoning, making this area one of the best ways of testing human-like intelligence. Namely, datasets based on games are common testbeds for machine learning systems (Fürnkranz, 2007). Usually, machine learning systems may be required to perform two different kinds of tasks (Fürnkranz, 1996). A first task is to learn a model that can be used to decide whether a move in a game is legal, or not. Having such a model is fundamental for the second task, where one wants to learn a winning strategy (Bain and Muggleton,

1994; Sadikov and Bratko, 2006). We focus on the the first task in this work. In order to acquire a meaningful representation of the set of rules describing a game, one can take advantage of the expressiveness of first-order logic and hence its ability to represent individuals, their properties and the relationship between them. Thus, using ILP methods it is possible - roughly speaking - to induce the game's rules written as a logic program, from a set of positive and negative examples and background knowledge.

Previous work has demonstrated the feasibility of using ILP to acquire a rule-based description of the rules of chess (Goodacre, 1996). However, to effectively learn chess theory, it is necessary to induce not only the top-legal concept of how a piece should legally move, but also subconcepts to help on that task. Consider, for example, Figure 4.1. In the first case, a king is in check and therefore its only legal moves are those which get it out of check. In the second case, a piece is playing the role of protector to the king (it is a pin). Before moving such a piece one should realize if the king continues to be protected, if the piece leaves its position. Those are intermediate concepts that must also be induced by the revision system. The authors of (Goodacre, 1996) employed hierarchical induction to learn a concept at each time, starting from the lowest level concept and incrementally adding the learned definitons to the final theory, until it reaches the ultimate goal: learning the concept of legal moves.

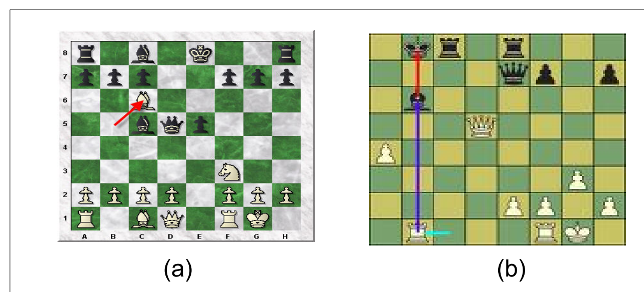

Figure 4.1: Visualization of situations when an ILP system should learn definitions for subconcepts. Figure (a) shows a board of chess with a checked king. Figure (b) shows a piece working as a pin.

On the other hand, game playing is a dynamic environment where games are always being updated, say, to be more challenging to the player, or to produce an easier and faster variant of the original game. In fact, popular games often have different regional versions, that may be considered *variants* or even a *new version* of the game. Consider, for example, the game of Chess, arguably, the most widely played board game in the world. It also has been a major game testbed for research on artificial intelligence and it has offered several challenges to the area. There are numerous chess variants, where we define *chess variant* as any game that is derived from, related to or inspired by chess, such that the capture of the enemy king is the primary objective (Pritchard, 2007). For instance, the *Shogi* game (Hooper and Whyld, 1992), is the most popular Japanese version of Chess. Although both games have similar rules and goal, they also have essential differences. For example, in Shogi a captured piece may change sides and return to the board [1], which is not allowed in Western Chess. Figure 4.2 shows boards of several chess variants. Ideally, if the rules of a variant of a game have been obtained, we would like to take advantage of them as a starting point to obtain the rules of a variant. However, such rules need to be modified in order to represent the particular aspects of the variant. In a game such as chess this is a complex task that may require addressing different board sizes, introducing or deleting new promotion and capture rules, and may require redefining the role of specific pieces in the game.

In this work, we address this problem as an instance of *Theory Revision from Examples* (Wrobel, 1996). In this case, theory revision is closely related to *Transfer Learning* (Thrun, 1995; Caruana, 1997), since the rules of international chess (the initial theory for the theory revision system) have been learned previously using ILP. Arguably, transfer learning is concerned about retaining and applying the knowledge learned in one or more tasks, to efficiently develop an effective hypothesis for a completely new task while theory revision deals with very related problems. For example, transfer learning may carry out a mapping between two different predicates but theory revision systems are not designed to perform such a task. However, after mapping one predicate to another it is usually necessary to change the definition of

[1] It is suggested that this innovative drop rule was inspired by the practice of 16th century mercenaries who switched loyalties when captured (Pritchard, 2007).

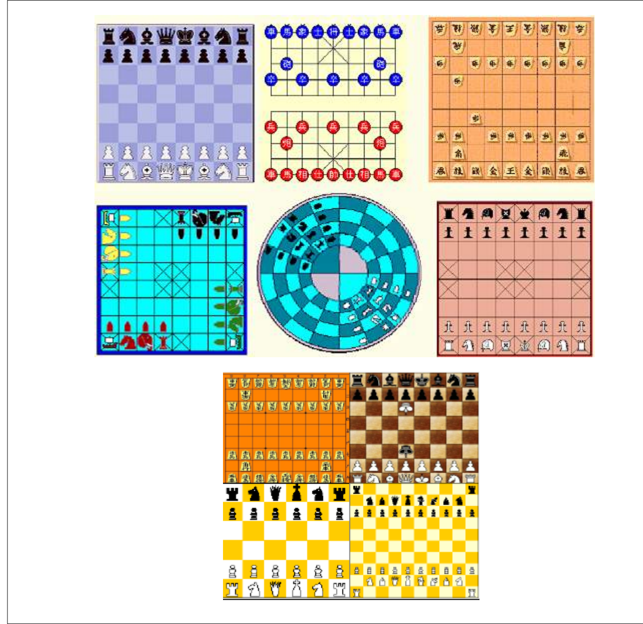

Figure 4.2: Boards of variants of chess. In the first row it is the international Chess, on its side it is the Xiangqi (Chinesse chess), followed by Shogi (Japanese Chess). Next row it one of the first chess games, the Chaturanga in the Hindu version, followed by the Chess in the Round game and a more modern version of Chaturanga, quite close to the International Chess. In the last group it is Shogi, Anti King Chess, Los Alamos and Grand chess, in this order.

the predicate, which is in fact a task of theory revision. Thus, theory revision may be seen as an important part of transfer learning systems.

We show that we can learn rules between different variants of the game of chess. Starting from YAVFORTE revision system explained in a previous chapter, we contribute with **(i)** a new strategy designed to *simplify* the initial theory by removing facts that will *not* be transferred between variants; **(ii)** support for *abduction*; and **(iii)** support for *negation as failure*. Experimental evaluation on real variants of chess shows that our technique can transfer between variants with smaller and larger boards, acquire unusual rules, and acquire different pieces. This chapter is organized as follows. First, we describe the components besides the revision system composing the framework for revising the rules of chess in section 4.2. Next, the modifications performed on YAVFORTE to support abduction and negation so that the problem is best addressed are described in section 4.3. Finally, we show

the effectiveness of our approach through experimental results on chess revision in section 4.4 and conclude the work in section 4.5. A reduced version of this chapter has been published in (Muggleton et al., 2009b) and (Muggleton et al., 2009a).

4.2 Revision Framework for Revising Rules of Chess to Turn Them in the Rules of Variants

The framework designed in this work for revising rules of chess so that laws of a variant of this game can be found is composed of five components, three of them are input components, one is the transformer component and the last one is the resulting component. The input components are (1) the initial theory, containing rules of international chess learned previously from examples and/or defined by an expert of the domain, (2) a set of examples reflecting the laws of the game's variant and responsible for pointing out where the initial theory differs from the domain of the variant and (3) a set of fundamental concepts supporting the provability of the initial theory. The transformer component is the theory revision system, which is responsible for modifying the initial theory according to the examples so that it reflects the rules of the variant of the game. Finally, the last component is the resulting revised theory, which ideally is a logic program capable of deciding whether a move in the game is legal or not, according to its governing rules. Figure 4.3 displays the components of the framework and how they cooperatively work to achieve the goal.

4.2.1 The Format of Chess Examples

As most ILP frameworks, theory revision refines an initial theory from a set of positive and negative examples. The Chess domain addressed in this work has as positive instances the legal moves allowed by the rules of the game. Consequently, the negative instances are the illegal moves, i.e., the moves not obeying the rules of the game. In order to represent the moves executed during a game, the dataset is composed of a set of simulated games up to a specified number of rounds. The moves are within a game aiming to represent castling and en-passant, which require the history of the games, and promotion, which require update of the board. In case of

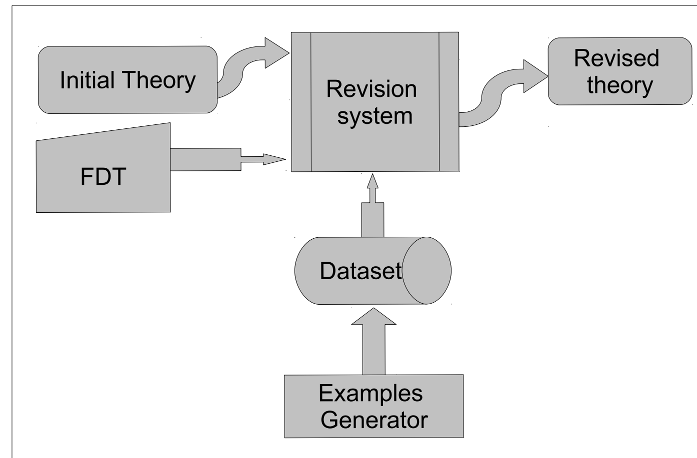

Figure 4.3: Components of the chess revision framework: Initial theory is going to be revised by the Revision system, using FDT and Dataset. Dataset is created from the Examples generator component.

castling, history is necessary to check whether the king and rook has already moved in previous rounds. To allow an en-passant it is necessary to check the immediately preceding move. Promotion requires to replace the promoted pawn with another piece in subsequent moves.

We take advantage of FORTE examples representation discussed in chapter 2.1, where an example obeys the format

$$Ground\, Instances\, of\, Target\, Predicate \leftarrow Conjunction\, of\, facts\, from\, the\, context$$

Thus, the ground instances of the target predicate are instances of legal (positive instances) and illegal (negative instances) moves and the facts from the context are the positions of the pieces related to each round (the board of the game). Each simulated game has its separate set of legal and illegal moves and set of positions of the piece during the game, in the format exhibited in Table 4.1.

The terms of the target predicate are the round of the move, with 1 as the first round, 2 the subsequent round, and so on, and the current and next *status* of the piece. The status is the name of the piece, its color and position, composed by File and Rank. For example, $move(9, pawn, white, c, 7, rook, white, c, 8)$ states that in round 9 a *white pawn* moved from $c, 7$ to $c, 8$ and is promoted to a *rook*, i.e, its next

Table 4.1: Format of one example in Chess dataset

```
Target Predicate:
    Positives:
        move(Round,Piece,Colour,File,Rank,NextPiece,
                NextColour,NextFile,NextRank),...
    Negatives:
        move(Round,Piece,Colour,File,Rank,NextPiece,
                NextColour,NextFile,NextRank),...
Context:
    board(Round,Piece,Colour,File,Rank),...
    out_board(Round,Piece,Colour,-1,-1),...
    out_board(Round,Piece,Colour,0,0),...
```

status is $rook, white, c, 8$. Similarly, the $move(5, bishop, black, c, 8, bishop, black, e, 5)$ states that in round 5 the *black bishop* moved from $c, 8$ to $e, 5$. The facts from the context represent the position of the pieces on the board at each round of the game, through the predicate $board/5$ and the pieces removed of the game by capturing or promotion moves, through the predicate $out_board/5$. In out_board/4 predicate, the two last terms are $-1, -1$ in case of a capture and $0, 0$ in case of a promotion.

The board setting is updated according to the legal move(s) performed on the previous round. Suppose, for example, the white bishop move above. There is a fact $board(5, bishop, black, c, 8)$ in that example and after the move, another fact is generated to represent the new position of the piece, namely $board(6, bishop, black, e, 5)$. Suppose there were a fact $board(5, pawn, white, e, 5)$ in the context of this same game. Thus, when the bishop has moved it captured a white pawn, which "produces" another fact $out_board(6, pawn, white, -1, -1)$ in the new setting of the board. The board facts for the first round are composed of the initial position of the pieces for the game and therefore they are the same for each example in the dataset. Figure 4.4 shows the initial board for the international chess and $board/5$ predicates representing it.

A move generator procedure is responsible for creating the dataset of simulated games, since we could not find any saved games for variants of chess. Basically, it starts from the initial board setting and successively chooses a piece from the

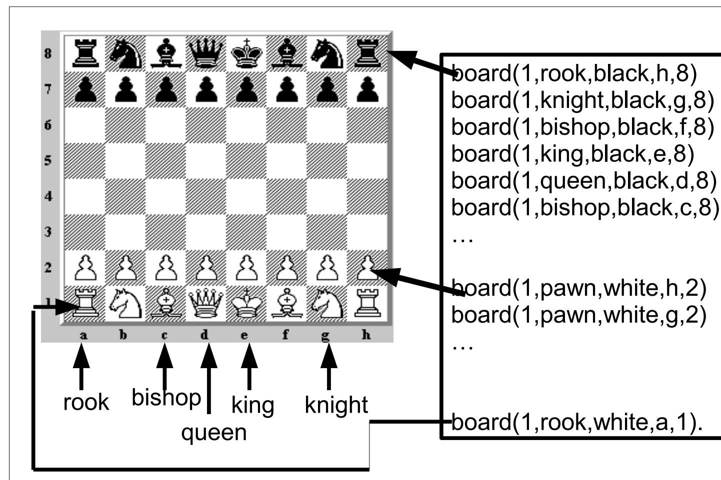

Figure 4.4: The initial setting of the board in international chess and atomic facts corresponding to it. Figure of the board is taken from $http://www.mark-weeks.com/aboutcom/bloa0000.htm$

current board, generating legal moves for it. After that it chooses at random one of the legal moves to be the positive example of the round. Then, it generates il illegal moves from pieces chosen at random from the current board to compose the set of negative examples of the round. Note that only one legal move is chosen because naturally each round of the game has one legal move. However, several possible illegal moves are selected, since they are not in fact performed. Moreover, the board must be updated based on the legal move, which does not happen from illegal moves, allowing us to have as many negative examples for each round as desired. The process continues up to the specified maximum number of rounds, respecting for each round the piece with the right to play. It may be also necessary to represent negative examples created from a non-existing board. This is the case of negative examples representing non-existing pieces or non-existing positions in the game. A variant of chess which does not use all the usual pieces or it uses a smaller board would require that. In this situation, a random board is created, specifically to represent such a negative example. Algorithm 4.1 shows the steps necessary to generate the set of examples. Table 4.2 exhibits part of one example in the dataset, extracted from a chess variant known as Gardner mini-chess (5X5 board).

Table 4.2: Part of one example in Mini-chess Dataset

```
example(
    % positive examples
      [move(p1,pawn,white,a,2,pawn,white,a,3),
      move(p2,pawn,black,c,4,pawn,black,c,3),
     % This is the black pawn that moved from c,4 to c,3 in round 2
      move(p4,pawn,black,c,3,pawn,black,b,2),...,
      move(p23,knight,white,c,3,knight,white,a,4),
     % A pawn is promoted to a knight
      move(p24,pawn,black,2,black,knight,c,1),
      move(p25,rook,white,d,1,rook,white,d,2)],
    % negative examples
      % promotion in an existing rank (therefore, from a non-existing board)
      [move(r94,pawn,bishop,white,b,7,bishop,white,b,8),
      move(p1,pawn,white,c,2,pawn,white,b,2),...,
      move(w84,queen,black,f,2,queen,black,b,2),...,
      move(p10,pawn,black,d,4,pawn,black,d,1),...,
      move(p25,king,white,e,1,king,white,f,1)],
   % facts from the context: board setting for each round
     board(p26,king,white,a,1),board(p26,knight,black,c,1),...,
     board(p25,knight,black,d,5),board(p25,rook,black,e,5),...,
     % A white pawn has been promoted to a knight
     % The fact such a pawn does not more belong to the board is
     % represented by out_board predicate
     out_board(p24,pawn,white,0,0),
     board(p24,knight,white,a,4),...,
     board(p23,pawn,white,a,3),board(p23,knight,white,c,3),...,
     board(p23,rook,black,e,5),...,
     % The black queen has been captured
     out_board(p25,queen,black,-1,-1),...,
     board(p5,rook,white,e,1),board(p5,queen,white,a,2),
     % Board updated after fourth round: a black pawn has moved to b,2
     board(p5,pawn,black,b,2),...,board(p5,pawn,black,e,4),...,
     board(p5,knight,black,d,5),board(p4,king,white,a,1),...,
     % Board updated after third round: the white queen has moved to a,2
     board(p4,queen,white,a,2),board(p4,pawn,white,b,2),...,
     board(p3,king,white,a,1),board(p3,queen,white,b,1),...,
     %Board updated after second round: a black pawn has moved to c,3
     board(p3,pawn,black,c,3),...,board(p2,pawn,white,e,2),...,
     %Board updated after first round: a white pawn has moved to a,3
     board(p2,pawn,white,a,3),board(p2,pawn,black,a,4),...,
     board(p1,bishop,white,c,1),board(p1,rook,white,e,1),
     board(p1,pawn,white,a,2),board(p1,pawn,white,b,2),...,
     board(p1,knight,black,d,5),board(p1,rook,black,e,5),
     % Non-existing boards are uptaded after a negative move
     board(r95,bishop,white,b,8),...,board(w9,rook,black,b,1)
```

Algorithm 4.1 Chess examples generator Algorithm

Input: The initial board setting CB, int d, representing the depth of each example/game, int il, number of negative examples for each round, number of examples n

Output: A dataset for a chess variant

1: $nex \leftarrow 1$
2: **while** $nex <= n$ **do**
3: $\quad round \leftarrow 1$
4: $\quad current_board \leftarrow CB$
5: $\quad color \leftarrow$ white
6: $\quad$ **while** $r <= d$ **do**
7: $\quad\quad piece \leftarrow$ choose a piece with color $color$ from $current_board$
8: $\quad\quad legal_moves \leftarrow$ generate legal moves for $piece \in CB$
9: $\quad\quad pos_move_round \leftarrow$ a move chosen at random from $legal_moves$
10: $\quad\quad neg_moves_round \leftarrow il$ illegal moves generated at random from CB and/or a non-existing board
11: $\quad\quad current_board \leftarrow$ update $current_board$ according to pos_move_round
12: $\quad\quad color \leftarrow$ black
13: $\quad\quad r++$
14: $\quad n++$

4.2.2 The Background Knowledge

In the chess revision problem, the initial theory describes the rules of the standard game of chess, which will be modified to state the rules of the variants, using appropriate examples. This theory is inspired on the one learned in (Goodacre, 1996) where hierarchical structured induction was employed in Progol system to learn clauses from the lowest level predicate (which does not have any other target predicate in the body) to the top-level predicate (the one which is not in the body of any other predicate). The resulting theory was approved by Professor Donald Michie, who was a world authority in computer chess research. To accomplish that, a set of examples were generated at each time as the target predicate of the respective level. Thus, different from her, the examples were not represented within a game, but isolated from each other.

The major differences between the theory exploited in the present work and the one learned in (Goodacre, 1996) concern the clauses describing special moves namely, castling, en-passant and promotion, since the authors of that work opted by not representing those moves. Additionally, we try as much as possible to avoid

negated literals with output variables to increase efficiency, as explained above, and because of that some rules were re-learned to obey this requirement.

Initial Theory

The initial theory, which must be revised to turn into the theory of the variant game rules, describes the rules for achieving a legal move following the rules of chess (Burg and Just, 1987). From a query about the legality of the move, the first step of the logic program is to inspect whether the opponent king is in check. If so, it is illegal to move the piece and the clause does not succeed. Otherwise, it proceeds trying to find a valid next position for the piece. In more detail:

1. Either the piece is a king, and:
   (a) It must be in a valid position (existing file and rank), and,
   (b) it must not go *next* to the opponent king, and,
   (c) it must not go into check, that is to a position threatened by an opponent piece and,
   (d) it cannot go to a position where there is already a piece of the same color, and then,
   (e) it can perform *castling* with a rook, if the castling conditions hold; or,
   (f) it moves one square in any direction to an*existing* position in the board.
2. Or the piece is *not* a king, and:
   (a) It must be in a valid position (existing file and rank), and,
   (b) it cannot move if its king is in double check (in this situation the only piece allowed to move is the king), and,
   (c) if it is protecting the king from a check (absolute pin) it can only move to a position where it continues protecting the king, or,
   (d) it can move to stop a check if the king is in simple check, either by capturing the threatening piece or by blocking the path between the threatening piece and the king, as long as it also respects conditions (g) and (h), or,

(e) it may perform an *attack*, only in case the intended position is already occupied by an opponent piece, by moving according to the direction it is allowed to, except for the pawn, or,

(f) if the piece is a pawn, and ((d) or (e)), it can perform an en-passant move; otherwise it captures one square diagonal, or,

(g) if the intended position is not occupied, the piece moves without capturing other piece;

(h) except for a knight, it must not pass over any other piece on the board while going to its next position;

(i) finally, it must move according to its basic move (bishops diagonally, rooks orthogonally, etc), to a valid position in the board (existing file and rank).

(j) If the piece is a pawn reaching the last rank it is promoted. The pawn can move 2 squares vertical forward only if this is its first move in the game.

Note that if the rules of the variant of the chess differ from the standard chess in any of the conditions above, the revision process must identify this through the examples and change those rules, so that they reflect the rules of the variant.

The pieces, files, ranks and color are represented in the initial theory as ground facts, where each fact is related to a piece, a file, a rank or a color allowed in the game of chess, as exhibited in Table 4.3.

Table 4.3: Ground facts representing pieces and colors in the game and file and ranks of the board, considering the interantional Chess

Pieces	Colors	Files	Ranks
piece(rook).		file(a). file(e).	rank(1). rank(5).
piece(knight).		file(b). file(f).	rank(2). rank(6).
piece(bishop). piece(pawn).	color(white).	file(c). file(g).	rank(3). rank(7).
piece(queen). piece(king).	color(black).	file(d). file(h).	rank(4). rank(8).

In order to avoid output variables in negated literals, some clauses were modified to provide the same definition but using only input variables. Suppose, for

example, the clause below explaining the concept of *check*, where the terms related to the piece checking the opponent king are unified with $(Piece, Colour_1, File_1, Rank_1)$.

$$check(Round, Piece, Colour_1, File_1, Rank_1, \texttt{king}, Colour_2, File_2, Rank_2) \leftarrow$$
$$nequal(Colour_1, Colour_2),$$
$$\%load\ the\ information\ about\ a\ piece\ on\ the\ board$$
$$board(Round, Piece, Colour_1, File_1, Rank_1),$$
$$attack(Round, Piece, Colour_1, File_1, Rank_1, \texttt{king}, Colour_2, File_2, Rank_2).$$

The clause above holds true if there is an opponent piece attacking the king; in this case $(Piece, Colour_1, File_1, Rank_1)$ states the opponent. If one simply needs to know whether the king is in check or not, the information regarding the attacking piece can be only local to the clause, i.e., it is not necessary to return them as answer substitution. Thus, a simplified predicate for deciding if the king is in check has a clause wherein all variables may be considered input:

$$check(Round, King, Colour_2, File_2, Rank_2) \leftarrow$$
$$check(BoardID, Piece, Colour, File_1, Rank_1, \texttt{king}, Colour_2, File_2, Rank_2).$$

In this case, the predicate *check*/5 has as input only the information about the king which might be in check, i.e., $Colour_2, File_2, Rank_2$ refers to the color and position of the possibly checked king.

Now, we can have $\texttt{not}(check(Round, King, Colour_2, File_2, Rank_2))$ in the body of a clause where all the terms are input variables, such as *load*, if we have a subgoal $board(Round, King, Colour_2, File_2, Rank_2)$ first. The same was done with other concepts in the theory appearing as negated literals in the body of some clause. Although this optimization was not required by the revision itself, it was an important useful step to achieve efficient construction of the bottom-clause.

The clauses representing special moves such as castling, en-passant and promotion were written by an expert and inserted by hand into the initial theory in appropriate clauses. Defining some part of the knowledge with the help of an expert

in the domain is usual in theory revision area.

Fundamental Domain Theory

Besides the initial theory, there is previous knowledge about the domain which is assumed as correct and therefore does not need to be revised. Fundamental Domain Theory contains definitions valid to the standard chess and also to the chess variants. During the revision process, this set of clauses is not modified and some of the clauses in the theory being revised need to use them so that they can be satisfied. After the revision process, the clauses in the FDT, together with the revised theory, will compose the theory of the chess variant. FDT is mainly responsible for keeping the definitions of fundamental concepts such as differences between positions (files or ranks), relations of equality, non-equality, greater than or less than between files and ranks, directions, among others. Below there is an example of a clause belonging to FDT, which defined the concept of *northdirection*.

%(b,2 is north from b,3)
$direction(File, Rank1, File, Rank2, north) \leftarrow less_thanRank1, Rank2).$

where *less_than* concept is defined as
$less_than(A, B) \leftarrow integer(A), integer(B), !, A < B.\ \%for\ ranks$
$less_than(A, B) \leftarrow name(A), name(B), A < B.\ \%for\ files$

The initial theory has 109 clauses with 42 intermediate predicates and the FDT 42 clauses. The work in (Goodacre, 1996) had 61 clauses in the BK and it learned 61 clauses. The difference in the total number of clauses is from the clauses that were added to avoid output variables in negated literals, as explained in the previous section, and from clauses representing special moves.

4.3 Modifying YAVFORTE to Acquire Rules of Chess Variants

Unfortunately, the revision algorithm described in chapter 3 cannot tackle the problem of revising between variants of chess. Analysis showed that chess generates a very large search space that could not be addressed well with uninformed search. Moreover, we must consider changes in the domain (such as differences in board size), that are not addressed well through the standard revision operators. We therefore propose a number of modifications to the current version of the system, described as follows.

4.3.1 Starting the Revision Process by Deleting Rules

In an attempt to decrease the complexity of the theory and consequently of the whole revision process, we introduce a first step of deletion of rules. This process is performed as a hill-climbing iterative procedure, where at each iteration the clauses used in proofs of negative examples are selected, each one is deleted in turn using the delete rule operator and the resulting theory is scored. The modified theory with best score is selected for the next step. The process finishes when no deletion is able to improve the score.

A similar algorithm was proposed to revise ProbLog programs (De Raedt et al., 2008b), where it has shown to be quite effective at finding minimal explanations. In our case, the goal would be to find a "common denominator" between the two different games. We found this procedure both reduces theory size and noise, namely when the target theory is a specialized version of the initial theory. On the other hand, it could implicate that the final revision is not the "minimally revised" one, since the proof of negative examples could be avoided through addition of few antecedents to the rules. However, the benefits on the decrease of runtime outweighs the possibility of returning a non-minimal revision.

After this step, the algorithm is executed as usual. The procedure for deleting rules is exhibited as Algorithm 4.2 and it is executed before the *line* 1 of the Algorithm 3.1. Note that the operator for deleting rules might be normally used again during the rest of the revision process (since it is one of the revision operators).

Algorithm 4.2 First Step for Deleting Rules

1: **repeat**
2: generate specialization revision points;
3: **for each** specialization revision point (a clause) **do**
4: delete clause;
5: update best revision found;
6: **if** best deletion improves the current score **then**
7: delete clause with best score;
8: **until** no deletion improves the theory;

4.3.2 Using abduction during the revision process

Abduction is concerned with finding explanations for observed facts, viewed as missing premises in an argument, from available knowledge deriving those facts (Flach and Kakas, 2000a). Usually, theory revision systems, including FORTE, use abduction when searching for generalization revision points, to locate faults in a theory and suggest repairs to it. Using abduction, a revision system determines a set of assumptions (atomic ground or existentially quantified formulae) that would allow the positive example to be proved. Consider, for example, the theory below, taken from (Mooney, 2000)

$p(X) : -r(X), q(X).$
$q(X) : -s(X), t(X).$

and the positive instance $p(a)$, with BK $r(a), s(a), v(a)$, unprovable by the current theory. The algorithm discovering revision points would find out by abduction that the instance is unprovable because the literal $t(a)$ fails. One of its suggestions to fix the theory would be to add a new definition for $t(X)$

We further benefit from abduction in three distinct situations of the revision process.

Intermediate Predicates Abduction Intermediate predicates are those ones appearing in the head of clauses and also in the body of others clauses, but there is neither example nor facts in the dataset corresponding to them. Suppose, for example the extracted piece of the chess theory in Table 4.4. The clause we show

in the table handles the case of a king moving by calling the clause defining how a king should legally move. Note that there are two clauses for doing this, one of them dealing with a king in check and another one to handle a move of a king not in check. Predicates *kingmove*/7, *kingmove_in_check*/7 and *kingmove_no_check*/7 are intermediate predicates, as they appear in the body of clauses and also in the head of others clauses. The instances in the dataset are of predicate *move*/4 only.

Table 4.4: An extracted piece of the chess theory, to exemplify the need of abduction when learning intermediate concepts

```
move(BoardID, king, Color, File1, Rank1, king, Color, File2, Rank2):-
        file(File1), rank(Rank1),
        file(File2), rank(Rank2),
        color(Color1), nequal(Co lour,Color1),
        equal(Piecek, king), % no constants in negated predicates
        % there is only one king of Color1 in the board
        board(BoardID, Piecek,Color1, File3, Rank3),
        % the king is not in check
        not(any_check(BoardID, Piecek, Color1, File3, Rank3)),
        % clause defining the legal move of a king
        kingmove(BoardID, king, Co lour, File1, Rank1, File2, Rank2).

kingmove(BoardID, king, Co lour, File1, Rank1, File2, Rank2):-
        kingmove_in_check(BoardID, king, Co lour, File1, Rank1, File2, Rank2).

kingmove(BoardID, king, Co lour, File1, Rank1, File2, Rank2):-
        kingmove_no_check(BoardID, king, Co lour, File1, Rank1, File2, Rank2).
```

It may be the case that the clause(s) defining intermediate predicates are wrongly defined. The error propagates to clauses containing literals with such predicate in their bodies, and so on. In this situation, such a literal is marked as a revision point, since it is in the path of the failing/successful proof of some positive/negative example. Suppose, for example there is no definition or a wrong definition for a king moving when it is in check, as in the example of Table 4.4.

In the first case, when the predicate is contributing to a positive example not to be proved, besides trying to modify clauses from where the intermediate predicate is called, the revision is going to propose the following modifications concerning the

predicate itself:

- Delete antecedents from the body of the clause defining an intermediate predicate, if such a clause exists;
- Add rules with the intermediate predicate in the head of the clause.

In the second case, an intermediate predicate is in the path of a successful proof of a negative example. Possible proposed modifications trying to solve this problem are going to be:

- Delete the rule with the intermediate predicate in the head.
- Add antecedents to the body of such a clause.

The second modification of each item concerns refining the clause by adding literals to its body. As we discussed in the previous chapter, this search space is composed of literals in the Bottom Clause, generated from a positive example covered by the clause, whose example predicate is the same as the one in the head of the clause. The problem is we have no examples for such intermediate predicates. In the chess theory, for example, the instances correspond to move predicates, but there is no example in the dataset for *kingmove*, *kingmove_in_check*, or *kingmove_no_check*.

Therefore, we need to tackle non-observation predicate learning (Muggleton and Bryant, 2000), where the concept being learned differs from that observed in the examples. We introduce *intermediate predicate abduction* in order to "fabricate" the required example. From a positive instance belonging to the answer set (relevant examples) of the intermediate clause, i.e, the proof of the instance includes the clause, we obtain an "intermediate instance" using the current theory and FDT. The procedure takes an example from the answer set together with the intermediate predicate and instantiates such a predicate to its first call encountered when attempting to prove the goal. The proof starts from the example and finds an instantiation for the specified intermediate predicate. After constructing the intermediate example, the bottom clause construction procedure is ready to run, followed by the refinement of the clause. The whole procedure can be visualized in Algorithm 4.3.

In Table 4.4, assume the predicate $kingmove_in_check/7$ is marked as a revision point and $move(p12, king, black, e, 9, king, black, b, 2)$ is one of the relevant examples. The intermediate instance $kingmove_in_check(p12, king, black, e, 9, b, 2)$ would be generated by the procedure, by first calling clause corresponding to $move/7$ predicate, next calling clause with $kingmove/7$ in its head and finally instantiating the intermediate instance.

FORTE also uses a similar procedure to generate extensional definitions for lower level recursive predicates, when revising theories with recursive clauses. In (Muggleton and Bryant, 2000) the procedure we employ here to abduct the intermediate instance is used to obtain "contra-positive examples" to the intermediate predicates and then construct a Bottom Clause for them.

Note that we can only reach an intermediate predicate from higher level clauses. In case there already exists a correct clause defining such a predicate, there is no need to further refine it and such a literal is a candidate to be included in the body of higher level clauses (clauses where the intermediate predicate is in the second term of a determination definition). However, if there is no clause defining this predicate, and also it does not belong to the body of a higher level clause, currently the revision process has no means of reaching this predicate and consequently propose the creation of a clause for it. This happens because, in order to make some modification in the provability of the example, the revision would have to be able to include the intermediate predicate in the body of some clause taking part in the proof of the example, and at the same time to create a clause for the intermediate predicate. Nowadays, the revision system is not capable of proposing modifications to more than one revision point at the same time. On the other hand, if there is a definition of the intermediate predicate, even if it is not correct for all required examples, but it solves the problems of some of them, the revision would be able to include such a predicate in the body of a clause. In a future iteration, the predicate should be marked as revision point and have its definition modified, so that the missing examples before become correctly proved. To sum up, although we do not escape from the limitations of inverse entailment (Yamamoto, 1997) with this procedure, it is enough to attend this class of requirement when revising a chess

theory.

To finish, it is worth mention there is great interest on learning concepts from non-observed predicates in ILP community, by handling two strategies of logical reasoning: induction and abduction (Flach and Kakas, 2000b; Inoue, 2001; Moyle, 2003; Ray et al., 2004).

Algorithm 4.3 Algorithm for Refining a Clause whose Head Corresponds to an Intermediate Predicate

Input: A theory T and FDT BK, a positive instance pi, a clause $C = l : -Body_0$, where l corresponds to an intermediate predicate, and $Body_0$ is a (possible empty) set of literals

Output: A possibly refined clause $C' = l : -Body_0 + Body_1$

1: $ipi \leftarrow$ Algorithm 4.4
2: $BC \leftarrow$ Bottom clause created from ipi, T and BK
3: $C' \leftarrow C$ refined with literals in BC
4: **return** C'

Algorithm 4.4 Algorithm for fabricating an intermediate instance

Input: A theory T and FDT BK, a positive instance pi, a clause $C = l : -Body_0$, where l corresponds to an intermediate predicate, and $Body_0$ is a (possible empty) set of literals

Output: An intermediate instance ipi

1: **repeat**
2: choose a clause to start the proof of pi
3: **while** $l \notin$ the derivation path or the proof does not finish, with a failure or a success **do**
4: continue to build the proof tree
5: **if** l is the atom in the leaf of the proof path **then**
6: try to prove $l : -Body_0$, accordingly unified
7: **if** it is possible to prove $l : -Body_0$ **then**
8: $ipi \leftarrow$ ground atom coming from the proof of l
9: **until** ipi is found

Abducting Atomic Facts to be Part of the Theory The simpler way of employing abduction in theory revision is to include in the theory assumptions that would allow positive examples become provable. Thus, the need for introducing an abducible predicate is identified during the search for generalization revision points. Suppose, for example, the extracted piece of the chess theory in Table 4.4. Assume the goal is to modify the theory so that it is possible to represent

a game of chess played in a larger board, say a 9X9 board. The positive instance $move(p12, [king], black, e, 8, e, 9)$ will fail, since it would not be possible to prove $rank(9)$.

We let any failing predicate to be a candidate to be abducted, once it obeys three additional requirements: (1) the predicate must have a `modeh` definition in modes declaration, so that it can become a clause whose body is empty, (2) there must have at least one `modeh` definition with only `constants` as its terms, and (3) the predicate must be defined as abducible. To abduct $rank(9)$, as necessarily demonstrated in the example above, a $modeh(1, rank(\#rank))$ would be required to be in modes definitions.

There is a maximum number of abducible predicates allowed in the theory, in order to prevent it to have several predicates maybe proving only one positive example. Note that this abduction is part of the add new rule operator, which acts on predicates, by creating an explanation for it. Because of that, an abducted predicate is only proposed as a revision if there is no way of creating a clause with such a predicate in its head. Thus, what really happens is: if the add new rule operator is unable of creating a clause defining the predicate, it is checked if it is abducible and if the maximum number of abducible predicates has not yet been reached. If the answer is affirmative, the operator looks for the `modeh` definition mentioned above and from it and an instance indicating the necessity of abduction, the atomic fact is created. As it is a proposed revision, the fact is only abducted if it brings an improvement to the current theory. Note that it is necessary to call Algorithm 4.4 so that the final atomic fact is found out. The algorithm performing this task is exhibited as Algorithm 4.5.

Note that if the abducible predicate becomes a faulting point in the theory, it may eventually be generalized/specialized in the next iterations.

Look-ahead abduction The last abduction approach addressed in this thesis is to use a strategy called here as *look-ahead abduction*, acting under the search for revision points. It works as follows. When searching for generalization revision points, it is assumed that faulty abducible predicates are true and the search continues,

Algorithm 4.5 Atomic Facts Abduction Algorithm

Input: Modes definitions M, abducible predicate $pred/N$, maximum number of abducible max_abd, Theory T, FDT BK, a positive instance pi

Output: An atomic fact $pred(t1, ..., tn)$, proposed to be included in T

1: **if** number of abducible predicates so far is less than max_abd **then**
2: $pred(t1, ..., tn) \leftarrow$ Algorithm 4.4
3: **if** $pred(t1, ..., tn)$ matches a `modeh` definition, where the terms are constants only **then**
4: **return** $pred(t1, ..., tn)$

looking for further revision points, possibly depending on the abducible predicate. In this way, it is possible to fix a clause with at least two problematic literals, since the first one failing is considered as proved.

Suppose, for example, we have the positive instance $a(1, 2)$, the negative instance $a(1, 3)$, the clause $a(X, Y) : -b(X, Z), c(X, Y, Z)$ and BK $d(1, 2, 4), d(1, 2, 5)$. The positive instance $a(1, 2)$ is unprovable and then generalization revision points must be found. When doing that, it is noticed the literal $b(1, Z)$ cannot be proved. Thus, originally, the search for revision points finishes, marking that clause and the literals $b(X, Z)$ and $a(X, Y)$ as generalization revision points. Notice that the revision points search procedure has no means to verify whether $c(1, 2, Z)$ could or not be proved, since the literal before it has already failed.

Although there are two revision points (the single clause and predicate $b/2$) it may be the case that no revision is able to fix the misclassification of the positive instance, without also making the negative instance provable. Considering the most common delete antecedent and add rule operators, note what the revision operators could propose:

1. Deleting either antecedent $b(X, Z)$ or $c(X, Y, Z)$: this does not make the positive instance become provable.

2. Creating the most general rule $a(X, Y)$ would make the negative instance provable. Only the most general rule could be created to this case.

3. Creating a rule capable of proving $b(1, _)$ does not solve the problem of the non-proof for $c(1, 2, _)$ and therefore does not bring any benefits to the theory.

However, if we assume $b/2$ as an abducible predicate during the search for

revision points, $c(X, Y, Z)$ is also marked as a generalization revision point and a rule could be created to explain $c(1, 2, _)$, say $c(X, Y, Z) : -d(X, Y, Z)$. Later, either $b(X, Y)$ would have to be removed from the clause or a definition would be created for it (possibly including the single assumption $b(1, 2)$).

For a more concrete example regarding the chess revision problem, consider the case of a game that has a different piece, say a *counselor*, that moves exactly like the bishop, but it cannot move backwards, only forward. Consider a simplified version of the move concept, in Table 4.5. The revision system cannot define the basic move for such a new piece, as it is required that the piece itself is known by the theory. However, if one abducts the literal *piece*(*counselor*) during the search for revision points, the system would be able to mark *basic_move* as revision point, and then propose a definition for it. The abducted literal *piece*(*counselor*) is going to be part of the proposed revision, besides the possibly created definition for basic_-move(BoardID, counselor, ...).

Table 4.5: An extracted piece of the chess theory, to exemplify the need of abducting predicates when searching for revision points

```
move(BoardID, Piece, Co lour, File1, Rank1, Piece, Co lour, File2, Rank2):-
        file(File1), rank(Rank1),
        file(File2), rank(Rank2),
        color(Co lour), piece(Piece),
        % the piece is on the board
        board(BoardID, Piece, Co lour, File1, Rank1),
        % any others necessary verifications
        ...
        basic_move(BoardID, Piece, Co lour, File1, Rank1, File2, Rank2).


        % how a bishop moves
basic_move(BoardID, bishop, Co lour, File1, Rank1, File2, Rank2) :-
        % abs_fdiff(File1, File2, Diff), abs_rdiff(Rank1, Rank2, Diff).
```

Thus, to outline the problem of more than one faulty literal in a clause, we take advantage of abduction and consider abducible predicates (up to a maximum number of predicates) in the theory under revision. To do so, if a failing abducible

literal is found during the search for generalization revision points (points failing on proving positive examples), it is assumed as correct and kept around in the revision point structure as a pending abducible predicate. Note that in this moment the literal must be an atomic fact. The search for revision points proceeds, by possibly including other abducible predicates as pending facts, until the proof tree reaches either a leaf or a failing literal not considered as abducible. In the first situation, the proposed revision is to include the abducible literals in the theory, as they were new rules, similar to what is done in the previous subsection. In the second case, the system marks the failing literal as a generalization revision point with pending abducible literals. When proposing modifications to such a point, the abducible predicates are considered as part of the theory. Eventually, in case the revision is chosen to be implemented, besides the generalization performed in the revision point, the system must also include the abducible predicates, otherwise the generalized revision point would continue failing.

Note that if the abducted predicate prevents some positive example of being proved or helps a negative example to be proved, the next iterations are going to mark them as generalization/specialization revision points. The process of obtaining abductive explanations and possibly generalizing them later is performed in (Moyle, 2003), but to learn clauses (completing definitions of background predicates) and not to revise them.

4.3.3 Using negated literals in the theory

YAVFORTE and FORTE_MBC are neither able to introduce negated literals in the body of the clause nor to revise negated literals. Negation is essential to elegantly model the chess problem, since we need to represent concepts such as *the king is* **not** *in check* or *a piece is* **not** *landing on another piece*, among others.

In order to add negated literals in body of clauses it is required that the Bottom Clause construction procedure is able to elicit them. Therefore, it is necessary to explicitly specify `modeb` and `determination` declarations for them. In the first case, the `modeb` declaration must be in the format:

$$\texttt{modeb}(RecallNumber, \texttt{not}(literal(Modes)))$$

stating that `not`(*literal*(...)) can appear in the body of some clause. Determination declaration must be in the format

$$\texttt{determination}(HeadPredicate/Arity, \texttt{not}(BodyPredicate/Arity))$$

stating that `not`(*BodyPredicate/Arity*) is allowed to be in the body of the clause whose head has *HeadPredicate/Arity*. In contrast to Aleph, we explicitly define the arity of the negated literal in determination definitions.

Inclusion of negation into logic programs is traditionally considered as a hard task since the incorporation of full logic negation tends to super-exponential time complexity of the prover. One of the most successful and widely alternative to full negation is the procedural approach of Negation as Failure (NAF) (Clark, 1978). One of the major drawbacks of NAF is that it cannot produce answer substitutions (new bindings) to negated query variables, requiring the negated literal is ground in order to be proved. If one wants more than a simple "yes" or "no" as answer, it is better not to apply negation as failure, since it may result in floundering of the goal (Marriott et al., 1990; Drabent, 1996). As a result, to use pure NAF inside the bottom clause construction procedure it is necessary to ensure there are no free variables in any `not`(*Goal*) that might be called. To do so, the mode type of all variables of negated literals must be of input so that it is guaranteed when `not`(*Goal*) is called, all terms are ground, since they should had been instantiated by previous literals.

The initial theory learned for the game of chess used in this work makes use of negated literals with output variables, since in some cases it is necessary to check out the negation of a concept. For example, it could be necessary to verify whether a king is not in check through the clause defining the check concept itself. Normally, such a clause should provide as output the piece responsible for putting the king in check. To address this problem the work presented in (Goodacre, 1996) made a special provision: a redefinition of `not` was introduced to the general proof which, after succeeding on the fail of the literal, it skolemises the variables so that they cannot be used again. We add the further requirement in the bottom clause construction procedure to guarantee the output variables of such literals are singleton (they appear only once in the clause) and therefore they must not be used as input variables of

others literals. Although we have re-learned some concepts so that output variables in negated literals become dispensable (subsection 4.2.2), we let the implementations above in the system, to attend a situation where this re-learn process would not be possible. Constant modes are not allowed in both works, since this would involve generating literals with all possible values, leading to a very inefficient process.

In case it is essential to bind variables in a negated query, the alternative most used to overcome the drawbacks of NAF is to employ constructive negation techniques (Chan, 1988; Drabent, 1995). It extends the NAF to handle non ground negative subgoals so that it is possible to construct new bindings for query variables. It works as follows: after running the positive version of the negated literal in the same way NAF does, the solution of the possibly non ground goal is collected as a disjunction and then this disjunction is negated to get a formula equivalent to the negative subgoal. The chess problem addressed here has not showed necessary to use constructive negation, since the initial theory does not need to bind variables from negated literals to answer some query. However, it would be nice to implement such an approach to the general case of negation in the revision process. We leave this question to future work.

In addition to inserting negated literals in the body of clauses, it may be the case that a negated literal is the culprit for the misclassification of examples. Remember from the laws of chess that a king cannot move next to the other king, in such a way that this last king can reach the former after a basic move. In the theory we tackle here, this is verified through a negated literal, as exemplified in line 6 of Table 4.6.

Now, suppose a variant of chess that is more restricted to obey this rule: a king cannot move to a position that is reachable by the other king after *two* basic moves. In this case, a positive example would be unprovable, since king_next_king/9 would succed, making its negation to fail. To fix this problem, one could modify the second clause in the table, so that it becomes more specific. One possible modification is to add a second basic_move/7 predicate in the body of the rule. Note that the clause is going to be specialized, even though we want to solve the problem of a positive example. Because the revision has been indicated by a negated literal, it

Table 4.6: An extracted piece of the chess theory, to exemplify the need of marking a negated literal as a revision point

```
kingmove1(BoardID, king, Color, File1, Rank1, File2, Rank2):-
        not(any_land_on(BoardID, king, Color, File1, Rank1, Color, File2, Rank2)),
        color(Color), nequal_color(Color,Color2),
        board(BoardID, king, Colour2, Filek, Rankk),
        basic_move(BoardID, king, Color, File1, Rank1, File2, Rank2),
        not(king_next_king(BoardID, king, Color2, Filek, Rankk, king, Color, File2, Rank2)).

king_next_king(BoardID, king, Color1, File1, Rank1, king, Color2, File2, Rank2):-
        basic_move(BoardID, king, Color1, File1, Rank1, File2, Rank2).
```

is necessary to reverse the proposed modifications: a clause corresponding to the negated literal must be specialized to fix the misclassification of positive examples, since this is going to make the clause not to be satisfied and therefore its negation succeed. Similarly, in situations where a negative example has been proved because the negation of a literal succeeds, it is necessary to generalize such a clause so that it can be satisified and its negation fails.

We thus introduced a procedure for handling a faulty negated literal during the revision process. Roughly speaking, if the negated literal is responsible for a failed proof of positive examples, it is treated as a specialization revision point. On the other hand, if the negated literal conducts to a proof of a negative example, it is treated as a generalization revision point. This is a preliminary attempt at introducing non-monotonic reasoning in YAVFORTE.

4.4 Experimental Results

Experimental methodology To show experimental results obtained with the framework discussed in the paper, we generated datasets for 3 different chess variants. The variants are described as follows, selected from (Pritchard, 2007).

- Gardner minichess. Minichess comprises a set of chess variants played with regular pieces and standard rules, but on smaller boards. There are games played on boards of size 3X3, 4X4, 4X5, 5X5, 5X6 and 6X6. The goal of this family of games is to make the game simpler and shorter than international

chess. Figure 4.5 shows several initial boards for minichess games. Here we focus on Gardner's minichess, which is the smallest chess game (5X5) in which all original chess pieces and legal moves are still used, including pawn double move, castling and en-passant.

- Reform chess, also known as Free capture chess differs from the international chess because either side may capture its own men, as well as the opponent's. Only the friendly king cannot be captured.

- Neunerschach chess is played on a board 9X9, with two extra pieces but removing the queen. The first one is called Marshall and moves like a queen. The second extra piece moves like a queen, but only two squares and it is named *Hausfrau*. The pieces are arranged in the board following the order: Rook-Knight-Knight-Marshall-King-Hausfrau-Bishop-Bishop-Rook.

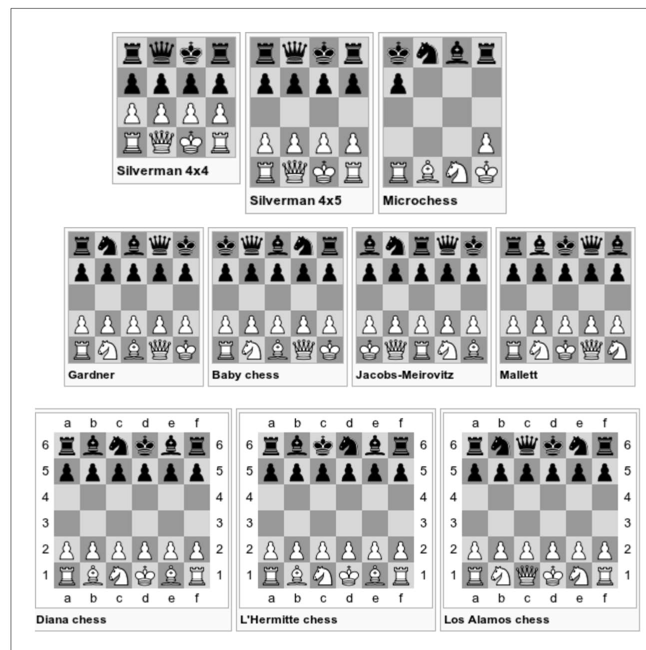

Figure 4.5: Initial boards of minichess games, taken from Wikipedia

We employed the framework described in this chapter to obtain theories describing each one of the above variants. We performed 5X2-fold cross validation and scored the revisions using f-measure. The datasets are arbitraly composed of 5

simulated game (example), where each round has 1 positive and 5 negative examples and the maximum round for each game is 25.

Next we present the evaluation measures obtained from each variant in Table 4.7.

Table 4.7: Evaluation of the revision on chess variants

Variant	# Pos. examples	# Neg. examples	# Facts in Context	# Initial clauses	# Final clauses	Initial Acc (%)	Final Acc (%)
Gardner minichess	97	500	3294	151	147	59	99.83
Reform chess	100	500	3360	151	154	93	100
Neunerschach chess	100	500	12651	151	160	87	100

Note that the variant with a smaller board had the size of the theory decreased but the others variants, which include all the rules of chess and also additional ones, had the size of the theory increased. We could say that the first case is a specialized version of the chess while the others are a more general version of the game. The number of clauses include the initial (revised) theory and the FDT, which is fixed. Notice that using only the initial theory can still achieve a good accuracy, as most rules are shared between different variants. On the other hand, accuracies were significantly improved through revision in every case, suggesting that the revision makes the dataset be correctly reflected.

The next subsection discusses specifically the revisions performed by the system in each variant together with remarks about them.

4.4.1 Discussions about the automatic revisions performed by the revision system

Gardner's Mini-chess

The system performed the following revisions on the initial theory to obtain a correct theory for Gardner minichess:

1. The *delete rule step* was able to remove the following clauses from the theory: $file(f)$, $file(g)$, $file(h)$, $rank(6)$, $rank(7)$, $rank(8)$. After that, any negative example coming from an invalid position or going to an invalid position becomes non-proved, since they do not attend one of the conditions 1.(a) , 1.(f),

2.(a) or 2.(h) of section 4.2.2.

2. The *add rule* generalization operator added the following clauses to the theory:
    - $basic_move(pawn, black, File, 4, File, 2)$, that allows the black pawn to move 2 squares. This is necessary because in the chess theory the black pawn moves 2 squares only from rank 7 (within the condition 2.(i)), which is not a valid rank in the Gardner Chess.
    - $promotion_zone(pawn, white, File, 5)$, that allows the white pawn be promoted when reaches the last rank of the Gardner mini-chess board.

Remarks about the revision/learning process in Gardner Mini-chess

- The final accuracy was on average 99.83%, since not all folds contained examples for promotion and black pawn moving 2 squares, as these moves are scarcely executed during the game.
    - Promotion occurs rarely in both chess and in Gardner mini-chess, since usually the pawn is captured before reaching the last rank.
    - The initial double moves of pawns are difficult in Gardner's smaller board, since, and differently from chess, the intended positions (rank 4 for white pawn and rank 2 for black pawn) are already occupied by opponent pieces and the pawn cannot capture a piece through its usual move.
- The best final theory is able to correctly classify all the moves of this variant, although it did not remove the clauses defining pawn double move from rank 7 to rank 5 and black pawn promotion when it reaches the rank 8, as to perform the former moves the pawn would have to be on rank 7. As previously stated, the clause defining this rank was removed from the theory and therefore it became an invalid rank. In this way, the condition 2.(a) of section 4.2.2 is not satisfied and the piece cannot move anyway. A post-pruning procedure should remove such useless clauses from the theory, so that the best final theory would perfectly correspond to the target theory of Gardner Mini-chess.

- We have tried to run Aleph and Progol 5 to find the rules of Gardner's minichess. As they do not revise a theory, we let the initial theory and FDT be the background knowledge. Aleph abduction procedure was not working properly. Progol, on the other hand, was able to abduct predicates basic_move and promotion_zone. However, it did not let such a predicate be in the final theory, as according to its evaluation function, there had been no improvement in score. The revision system is more fortunate in this question, since it focuses on the misclassified examples to propose revisions. Progol could neither delete rules, as this is not one of its refinements operators.

Reform chess: Unusual capture rule

The system performed the following revisions on the initial theory:

1. According to condition 1.(e) of section 4.2.2, the king must not go to a position where there is already a friendly piece. This is declared through subgoals of the form `not`($land_on(..,Colour,...,Colour,...)$) in the clauses defining how a king may move. As free capture chess allows the king to capture a friendly piece, the *delete antecedent* operator removed these literals from their respective clauses.

2. In the original theory, a piece may move to an occupied position only if an opponent piece is there. This is defined as a move by the attack concept. In Reform chess the piece is also allowed to attack a friendly piece. Because of that, the revision chosen to be implemented in most of the folds is proposed by the *add rule* operator. The main literals in the body of such a rule, allows an attack to a piece of the same color, only if such a piece is not a king. Note that the original rule defining the attack concept remains on the theory.

3. Progol does not proposed to create new rules for the same predicate as the revision. Instead, it tried to learn a new definition of the top-level move predicate, without success.

Unusual pieces and larger board: Neunerschach

The system performed the following revisions on the initial theory:

1. The delete rule step discussed in section 4.3.1 removed the clause $piece$(`queen`) from the theory;

2. Abduction when searching revision points (third introduced type) added the facts $piece$(`marshall`) and $piece$(`hausfrau`) to the theory and, just after that,

3. From the rules defining the basic moves of the queen, the add rule operator created rules for the `marshall` by replacing the constant `queen` from the head of such rule by `marshall`. This was done by first treating the constants in the clause, substituting them by equality predicates. FORTE always does that before searching for revision points. Then, the literal was deleted and the constant `marschall` was made to replace `queen`;

4. The add new rule operator included the following new rules in the theory

    - Rules defining the basic move of `hausfrau`, so that this piece can move diagonally and orthogonally, but only two squares;
    - New ground clauses, produced by abduction of the second type, to make the last file and rank valid, namely, $file$(`i`) and $rank$(9).

Remarks about the revision process in Neunerschach chess Note that without the abduction procedure it would be difficult for the revision process to induce the rules for the new pieces, since first of all, the piece must be valid to match condition 2.(a) of section 4.2.2 and only after that the piece can try to move. Thus, only including the clauses defining the pieces would not bring any benefit to the theory, since there were not any clauses defining how such pieces should move. Additionally, as the pieces did not exist on the theory, the search for revision points procedure would not be able to reach the literals defining the basic moves of the pieces and then identify them as revision points. Even if it could, defining the basic moves of the clauses is not enough for they to move, since they must be valid in the theory.

The move generator procedure was not able to generate games with promotion cases for this dataset, probably due to the size of the board X maximum number

of rounds in the game. Thus, the revision process failed on correcting the rules of promotion, which would allow the pawn to be promoted to the new pieces and the white pawn be promoted in rank 9, instead of in the rank 8. We expect that using games with a larger number of rounds will allow us to represent them, but at the cost of much larger datasets.

Finally, Progol was able of proposing abduction of predicates file/1, rank/1 and piece/1, but it could not create clauses defining basic moves of the new pieces.

4.5 Conclusions

We presented a framework for learning variants of a game through automatic theory revision from examples. The framework is composed of a Theory Revision system, the chess initial rules (expressed as the initial theory which is allowed to be modified) and fundamental domain theory (assumed to be correct) and a move generator for obtaining the examples. We described the modifications performed on the revision process to best address the revision of chess problem, including (1) the introduction of an initial step for deleting rules responsible for misclassified negative examples, (2) the use of abduction in three different moments and (3) the use of negation. The experimental results encompassed 3 variants of chess, ranging from a specialized version of chess (minichess) to a more general version of chess (including a larger board and new pieces). The revision was able to return final theories correctly describing most of the rules of the variants. The missing cases were due to the lack of examples of rare events during a game, such as the promotion. Also, the final theory would benefit from a post pruning procedure to remove clauses that have become useless after the revision.

We are aware that the datasets were generated in a quite arbitrary way, since the number of examples and depth of the games were chosen according to the need of generating situations necessary for the revision. A better experimental setting should include several datasets of at least varying sizes. Notice however that our primary goal was to demonstrate the capability of the revision system acquiring the rules of variants of the game, using the rules of the traditional game as starting point. Also, we would like to show that it was necessary to change the base revision

system to achieve that. In this way, new issues were introduced in the revision system that can also be useful for others domains besides chess.

Thus, as future work, we intend to further experiment the framework with datasets varying on the number of maximum rounds and simulated games. We would like to apply the framework to other more complex variants, namely regional variants, such as Shogi. Shogi and several others variants of chess require the knowledge of an entirely new concept. The *Einstein chess* variant, for example, has the concept of demotion, where each time a piece moves without capturing it is "demoted" to a less valuable piece. This concept does not exist in international chess and if one would like to obtain the theory for this variant, literals for demotion would have to be included in the language by hand. This is not the best scenario, since it is assumed the previous knowledge concerns the situations occurring in traditional chess only. Therefore, the revision system would greatly benefit from predicate invention operators (Muggleton and Buntine, 1988).

A more ambitious future work is learning playing strategies (Bain and Muggleton, 1994). In this case, we would like to obtain strategies for playing the variants of chess from strategies of chess. To do that we believe we would benefit from probabilistic theory revision, so that uncertainty about the strategies could be represented.

Finally, we would like to apply theory revision to others tasks involving transfer learning, such that the domains share the same predicates language. We believe in this way it would not be necessary to map between predicates of one domain to predicates of another domain.

Chapter 5

Stochastic Local Search

Searching over large spaces is a recurrent problem in Computer Science. In order to get good hypotheses while still keeping the search feasible, one may take advantage of local search algorithms, which start generating candidate hypothesis at some location of the search space and afterward moving from the present location to a neighbouring location in the search space. Each location has a relatively small numbers of neighbours and each move is determined by a decision based on local knowledge (Hoos and Stützle, 2005). In this way, they abandon completeness to gain efficiency.

It is also possible to further improve efficiency and also escape from local optima by making use of randomised choices when generating or selecting candidates in the search space of a problem, through the *Stochastic Local Search Algorithms*(SLS). One major motivation and successful application of SLS has been in satisfiability checking of propositional formulae, namely through the well-known GSAT (Selman et al., 1992) and WalkSAT (Selman et al., 1996) algorithms. A large number of tasks in areas such as planning, scheduling and constraint solving can be encoded as a satisfiability problem, and empirical observations show that SLS often can substantially improve their efficiency (Chisholm and Tadepalli, 2002; Rückert and Kramer, 2003).

Randomisation may also be used to improve other search strategies, such as *backtracking* search. For example, Rapid Randomised Restarts (RRR) introduces a stochastic element intro backtrack-style search in order to introduce restart search from scratch if we are not making progress (Gomes et al., 1998; Gomes et al., 2000).

This led to an interest in applying such techniques on data-mining applications, and more specifically on multi-relational data-mining. Initial work on the area has indeed shown promising results for stochastic techniques when learning theories from scratch in ILP systems (Paes et al., 2006b; Železný et al., 2006). For two very different algorithms, results showed very substantial improvements in efficiency, with little or no cost in accuracy.

Motivated by these work, this thesis also contributes in the development of SLS methods when revising first-order logical theories. Thus, in order to ground the contributions of next chapter, this chapter presents an overview of SLS algorithms and SLS methods employed in Inductive Logic Programming. The chapter starts by describing standard SLS methods in section 5.1. Then, well-known and vastly used SLS algorithms, such as GSAT and WalkSAT, are reviewed in section 5.2. Finally, in section 5.3, SLS algorithms designed to efficiently learn first-order logic theories are discussed. In this last section it is included one technique developed with our contribution, where first-order logic theories are learned using propositionalization combined with a SLS algorithm (Paes et al., 2006b).

5.1 Stochastic Local Search Methods

The key ideas of the search process performed by a Stochastic Local Search Algorithm are as follows.

1. Initialisation: An initial candidate solution is selected, usually by generating a candidate at random;

2. Move step: Iteratively, the process (at random) decides to move from the present candidate solution to a local neighbouring candidate solution, usually (but not always) considering a function to evaluate the neighbours.

3. Stop criteria: The process is finished when it attends a termination criteria, which could be a maximum number of iterations or a solution has been found.

Suppose for example, a *Stochastic Hill Climbing* (or Iterative Descent) strategy (Russell and Norvig, 2010). It starts from a randomly selected point in the

search space (initialisation) and tries to improve the current candidate solution by choosing with uniform probability distribution a neighbour of the current candidate (move), but requiring the value of the evaluation function is improved. The process finishes when none of the neighbours improves the evaluation function (stop criteria). A *Greedy Stochastic Hill Climbing* strategy (or Discrete Gradient Descent) would try to perform the best move by choosing uniformly the next candidate solution in the set of maximally improving neighbours (the best possible improvement). As such a method requires a complete evaluation of all neighbours in each iteration, one may prefer a *First Improvement* neighbour selection strategy, which moves to the first neighbour encountered improving the value of the evaluation function.

5.1.1 SLS Methods Allowing Worsening Steps

It is easy to note that the performance of a SLS algorithm (as it also occurs in other local search methods) strongly depends on the size of the neighbourhood. The larger is the neighbourhood, the more potentially better candidate solutions it contains. In fact, the ideal case is to have an exact neighbourhood: the neighbourhood relation for which any locally optimal candidate solution is guaranteed to be the globally optimal solution. Obviously, taking into account an exact neighbourhood relation prevents the search method to be stuck in very low-quality local optima. However, it is also much more expensive to determine search improving steps in exact neighbourhoods.

It is possible to escape from local optima and avoid an expensive search considering a fairly simple neighbourhood and allowing the search strategy to perform worsening steps. Usually, a strategy following this idea alternates with a fixed frequency between selecting an improving neighbour and selecting a neighbour at random. In order to avoid cycles, which may happen if the random walk is undone in subsequent improvement steps, the algorithm can probabilistically decide in each step whether to apply an improvement step or a random walk step. The family of algorithms following this strategy is called *Randomised Iterative Improvement (RII)* and its top level search step is exhibited as Algorithm 5.1. A RII algorithm does not terminate as soon as a local optima is encountered. Instead it may stop the execution when it reaches a number of iterations or when a number of search steps has

been performed without making progress in improvement. It is proved that when the search process runs long enough, eventually an optimal solution to any problem instance is found with arbitrarily high probability (Hoos and Stützle, 2005).

This method had been successfully applied to SAT problems, through the WalkSAT algorithm (Selman et al., 1996), which we will discuss more thoroughly in next sections.

Algorithm 5.1 Top-level Step Function Performed by an Algorithm using Randomised Iterative Improvement method (Hoos and Stützle, 2005)

Input: problem instance π, candidate solution s, walk probability wp
Output: candidate solution s'
1: $u \leftarrow$ random(0,1) # returns a number between zero and one
2: **if** u $\leq$ wp **then**
3: $s' \leftarrow$ random_walk_step(π, s)
4: **else**
5: $s' \leftarrow$ improvement_step(π, s)
6: **return** the result of step 3

Another mechanism for allowing worsening steps it to accept a worsening step depending on the deterioration in the evaluation function value, i.e., the worse a step is, the less likely it would be performed. The algorithms following such strategy compose the family of *Probabilistic Iterative Improvement (PII)*. In each step of the search process a PII algorithm selects a neighbour according to a given function $p(g, s)$, which determines a probability distribution over neighbouring candidate solutions of s based on their evaluation function values g.

A vastly used strategy, which is closely related to PII is the *Simulated Annealing (SA)* method (Kirkpatrick et al., 1983; Geman and Geman, 1984; Cerny, 1985; Laarhoven and Arts, 1987) (Top-level algorithm in 5.2). SA starts from a random initial solution s and in each iteration of the search a neighbour s' of s is selected at random. Usually, the search makes the decision of moving to s' or staying in s based on a value T, which is adjusted at each step t by an *scheduling function*. Standard SA always accepts the candidate s' in case it has a score better than s. When it does not happen, s' is accepted with a probability calculated as the exponential of difference between both scores divided by T. Thus, the worse is the move and the less is the value T, the less exponentially is the probability. Thus, bad moves are

chosen more frequently at the beginning of the search, when the value of T is high, and they become more difficult to be accepted as the value of T decreases. It has been proved that if the value of T is reduced slowly enough, the algorithm finds a global optima with probability close to 1 (Russell and Norvig, 2010).

Algorithm 5.2 Simulated Annealing algorithm (Geman and Geman, 1984)

Input: Integer number k and $limit$; a float number lam
Output: a solution s

1: $s \leftarrow$ random generated candidate solution
2: **for** t from 1 to ∞ **do**
3: $\quad T \leftarrow$ scheduling$(t, k, limit, lam)$
4: $\quad$ **if** $T == 0.0$ **then**
5: $\quad\quad$ **return** s
6: $\quad s' \leftarrow$ a neighbour of s, chosen at random from the neighbourhood relation
7: $\quad \Delta\ E \leftarrow$ score(s') - score(s)
8: $\quad$ **if** $\Delta\ E > 0$ **then**
9: $\quad\quad s \leftarrow s'$
10: $\quad$ **else**
11: $\quad\quad s \leftarrow s'$ only with probability $e^{\Delta\ E/T}$

5.1.2 Rapid Random Restarts

Another way of escaping from local optima is to simply restart the search algorithm whenever it reaches a local minimum (maximum). Such a strategy works reasonably well when the number of local optima is rather small or reinitialising the process is not very costly. The development of this approach was motivated by the recognised variability in performance found in combinatorial search methods (Gomes et al., 1998; Gomes et al., 2000), such as satisfaction constraint problems. Often, the cost distributions of a complete backtracking search have very long tails and an average erratic behaviour. Search algorithms with *RRR* include a stochastic component in backtracking style search. The key idea is to restart the search, possibly from another random initial seed, if it is not making any progress after a number of tries. To do so, a *cutoff* specifies a number of attempts of finding the best solution before restarting from another point.

A simple example of RRR strategy applied to a hill-climbing algorithm conducts a series of improving moves from a random initial candidate solution. The

algorithm is complete with probability close to 1 for the simple reason that it will eventually generate the goal solution as the initial solution. If each iteration of the hill climbing has a probability p of success, the expected number of restarts is $1/p$ (Russell and Norvig, 2010).

5.1.3 Other SLS approaches

Others SLS methods exist.*ILS* (Lourenćo et al., 2002) combines two types of SLS methods, one in each step. One step reaches local optima as fast as possible and the other step escapes from local optima. At each iteration, Iterated Local Search (ILS) first applies a perturbation to the current candidate solution s, yielding a modified candidate solution s'. A local search is performed from s' until a local optimum s'' is obtained. Finally, an acceptance criteria decides if the next candidate solution is s' or s''. *Greedy Randomised Adaptive Search Procedures (GRASP)* (Feo and Resende, 1995) randomise the construction method of candidate solutions such that it can generate a large number of good starting points for a local search procedure. Evolutionary algorithms are a large and diverse class of stochastic search methods strongly inspired by models of the natural evolution of biological species (Bäck, 1996). Generally speaking, evolutionary algorithms such as genetic algorithms (Mitchell, 1996) start with a set of candidate solutions and repeatedly apply three genetic operators, namely selection, mutation and recombination, replacing partially or completely the current population by a new set of candidate solutions. For the knowledge of the much many others SLS approaches and a better understanding of the methods briefly described in this section, we refer the reader to (Hoos and Stützle, 2005).

5.2 Stochastic Local Search Algorithms for Satisfiability Checking of Propositional Formulae

Stochastic local search has been used since the early nineties to solve hard combinatorial search problems, starting from the seminal algorithm published independently by Selman (Selman et al., 1992) and Gu (Gu, 1992). This SLS algorithm was able to solve hard satisfiability problems in only a fraction of the time required by the most sophisticated complete algorithms. The satisfiability problem in propositional

logic (SAT) must decide whether there exists an assignment of truth values to the variables of a formula F under which F evaluates to true (a model of F). SLS are among the most successful methods for solving the search variant of SAT, i.e., to find models for a given formula rather than to decide if such a model exists. In the following we discuss the seminal GSAT algorithm (Selman et al., 1992) and its most arguably prominent derivative, the WalkSAT algorithm (Selman et al., 1996).

5.2.1 The GSAT Algorithm

GSAT (Selman et al., 1992) original algorithm is a SAT solver based on a greedy hill climbing procedure with a stochastic component. It is based on a one-flip-variable neighbourhood in the space of all complete truth values assignments of a given formula, written in conjunction normal form (CNF), i.e., a conjunction of clauses, where each clause is a disjunction of literals. Thus, two variable assignments are neighbours iff they differ in the truth assignment of exactly one variable. GSAT uses an evaluation function $g(F, va)$ that maps each variable assignment va to the number of clauses of the given formula unsatisfied under va. At each iteration of the algorithm, $g(F, va)$ must be minimised by flipping the value of one variable, since the goal is to find a model of F, therefore evaluated to zero under $g(F, va)$. The variable to be flipped is selected at random from the neighbourhood of current candidate solution minimising the number of unsatisfied clauses (Hoos and Stützle, 2005).

Algorithm 5.3 presents the basic GSAT algorithm. It has two nested loops, with the inner loop starting from a randomly chosen truth assignment of the variables in CNF formula F. Then, it iteratively flips the variable resulting in a maximal decrease in the number of unsatisfied clauses. If there is a tie, the variable to be flipped is chosen at random from the set of variables improving the score at most. The inner loop continues until it finds an assignment of variables satisfying F or when it reaches a user defined number of steps. In case no model is found after a maximum number of steps, GSAT reinitialises the search at another randomly chosen truth assignment of F. This is strictly necessary, since the inner loop gets easily stuck in local minima. The outer loop follows trying to find a model of F

until a user defined number of tries. After the maximum number of tries without finding a solution, the algorithm ends with "no solution found".

Algorithm 5.3 GSAT Algorithm (Selman et al., 1992)

Input: Positive integers $maxFlips$ and $maxTries$; a CNF formula F
Output: model of F or "no solution found"

1: **for** try from 1 to $maxTries$ **do**
2: $va \leftarrow$ a randomly generated assignment of the variables in formula F
3: **for** $step$ from 1 to $maxSteps$ **do**
4: **if** va satisfies F **then**
5: **return** va
6: $v \leftarrow$ randomly selected variable flipping which minimises the number of unsatisfied clauses
7: $t \leftarrow t$ with v flipped
8: **return** "no solution found"

Due to the greedy hill climbing nature of GSAT, it suffers from a severe stagnation behaviour, getting easily stuck in local optima for any fixed number of restarts. Thus, GSAT has been extended into to other search strategies in order to improve its performance. One of the most prominent is the GWSAT algorithm (Selman and Kautz, 1993) which decides at each step with a fixed probability np whether to do a standard GSAT step or to flip a variable selected uniformly at random from the set of all variables occurring in unsatisfied clauses. The probability np is called *walk probability, noise setting or noise level.* The WalkSAT algorithm, which we discuss next is derived from GWSAT.

5.2.2 The WalkSAT Algorithm

WalkSAT (Selman et al., 1994; Selman et al., 1996; Selman et al., 1997) changes GSAT based algorithms mainly by considering only a dynamically determined subset of the GSAT static neighbourhood relation. This is effectively done by first considering only variables occurring in unsatisfied clauses. Then, in order to find the next assignment, the variable to be flipped is selected from that set in two steps. In the first step, a clause c, which is unsatisfied under the current assignment of truth values, is selected at random. Next, in a second step, the new assignment results by flipping one of the variables appearing in c. This general procedure of the

WalkSAT architecture is exhibited as Algorithm 5.4. Note that as GSAT, WalkSAT starts from a randomly generated assignment of the variables in initial formula. Also as GSAT, it considers a maximum number of tries and a maximum number of steps in order to find the solution.

Algorithm 5.4 WalkSAT General Algorithm (Selman et al., 1994)

Input: Positive integers $maxFlips$ and $maxTries$; a CNF formula F; a heuristic function hf
Output: model of F or "no solution found"

```
1:  for try from 1 to maxTries do
2:      va ← a randomly generated assignment of the variables in formula F
3:      for step from 1 to maxSteps do
4:          if va satisfies F then
5:              return va
6:          unsat ← the set of all unsatisfied clauses under va
7:          c ← a randomly selected clause from unsat
8:          v ← a variable selected from c according to the heuristic function hf
9:          t ← t with v flipped
10: return "no solution found"
```

The general procedure requires a heuristic function as input to decide which variable is going to be selected, in line 8 of the algorithm. The most commonly used heuristic function applied into the WalkSAT architecture first scores each variable v by counting the number of currently satisfied clauses that will become unsatisfied by flipping the variable. Then, it tries to perform a *zero damage step*: if there is a variable with score equal to zero, that is, if the the clause c becomes satisfied by flipping the variable v without damaging another clause, then v is flipped (if there is more than one variable in this situation, one of them is chosen at random). In case it is not possible to follow the zero damage step, WalkSAT must decide which step to follow: a *random walk step* or a *greedy step.* The decision is taken considering a walk probability wp as follows:

- With a certain probability wp one of the variables from the clause c is selected at random to be flipped (random walk step);
- With probability $1 - wp$ the variable with the minimal score calculated as above is selected to be flipped (greedy step)

This heuristic function is presented as Algorithm 5.5.

Algorithm 5.5 Most Used Heuristic Function into WalkSAT Architecture

Input: va a variable assignment of a formula F; a clause c; wp, a walk probability
Output: v a variable to be flipped

1: $scores \leftarrow$ for each variable in c, the number of clauses that are currently satisfied, but become unsatisfied if the variable is flipped
2: **if** $min(scores) = 0$ **then**
3: $v \leftarrow$ a random variable from c whose score is $min(score)$
4: **else**
5: **with** probability wp **do**
6: $v \leftarrow$ a random variable from c
7: **otherwise**
8: $v \leftarrow$ a random variable from c whose score is $min(score)$

Following the success of GSAT/WalkSAT based algorithms to check the satisfiability of propositional formulae, a large number of randomised strategies were derived from them to solve tasks in areas such as planning, scheduling and constraint solving. Empirical observations show that SLS often can substantially improve their efficiency (Chisholm and Tadepalli, 2002; Rückert and Kramer, 2003; Rückert and Kramer, 2004).

5.3 Stochastic Local Search in ILP

Most Inductive Logic Programming algorithms perform search on a large search space of possible clauses, leading to huge time and storage requirements and urging for clever search strategies (Page and Srinivasan, 2003). It is therefore unsurprising that research on stochastic search has taken place since early ILP days (Kovacic et al., 1992). Many of the ILP algorithms indeed include a limited amount of stochastic search. As an example, GOLEM system randomly select examples as *seeds* to start their search (Muggleton and Feng, 1990). Next we present recent work on stochastic search in ILP.

5.3.1 Stochastic Local Search for Searching the Space of Individual Clauses

A recent study in ILP implemented and evaluated the performance of several randomisation strategies in the ILP system Aleph (Železný et al., 2002; Železný et al., 2004; Železný et al., 2006), using Aleph deterministic general-to-specific search as reference. Aleph runs a covering algorithm which obtain hypotheses inducing clauses one by one, until all the positive examples are covered. Roughly, Aleph's covering procedure is composed of two nested loops. The inner loop starts with the generation of the bottom clause from a seed example. Then, a clause with the best score is generated by adding antecedents from the bottom clause. The outer loop of the covering algorithm removes already covered positive examples from the set of examples and the inner loop restarts with this new set of examples. The procedure continues until there are no move not-covered positive examples or it is not possible anymore to generate clauses obeying the setting of parameters defined by the user. Note that the inner iteration returns a single clause generated independently from the clauses previously added to the theory. Thus, the stochastic strategies developed in that study were framed in terms of a single clause search algorithm.

They designed four randomised restart strategies to search the ILP subsumption lattice, namely (1) A simple randomised search strategy (RTD); (2) A Rapid Random Restart strategy (RRR); (3) A GSAT based strategy and (4) a WalkSAT based strategy. The randomised strategies differ on how to choose the saturated example, on which clause to start from, on which clause to try next, on whether to do greedy or full search, and on whether to do bidirectional refinement or not, following the main properties:

1. the saturant example is chosen at random in all randomised algorithms, instead of being the first positive example as in the deterministic reference strategy;

2. the search starts from a clause selected with uniform probability from the set of allowable clauses, except for the RTD search which which starts from the most general definite clause (Srinivasan, 2000);

3. GSAT and WalkSAT strategies update the list of possible modifications in the

current hypothesis greedily, i.e., only the newly explored nodes are retained, whereas RRR and RTD maintain a list of all elements.

4. the selection of next clause follows a random choice in WalkSAT and in RTD, according to the following criteria: with probability 0.5 the clause with the highest score is chosen at random and otherwise a random clause is chosen with probability proportional to its score. The others strategies choose the clause with the highest score.

5. GSAT, WalkSAT and RRR perform bidirectional refinement, combining specialisation and generalisation, instead of only performing specialisations of the clause being refined.

6. All strategies include restarts based on a cutoff parameter, defined as the maximum number of clauses evaluated on any single restart.

It was observed that if a near-to-optimal value of the cutoff parameter (the number of clauses examined before the search is restarted) is used, then the mean search cost (measured by the total number of clauses explored rather than by cpu time) may be decreased by several orders of magnitude compared to a deterministic non-restarted search. It was also observed that differences between the tested randomised methods were rather insignificant.

There are others approaches of SLS methods for searching the space of candidate clauses. Particularly, Muggleton and Tamaddoni-Nezhad have been conducting research based on the stochastic search performed by genetic algorithms (Tamaddoni-Nezhad and Muggleton, 2000; Muggleton and Tamaddoni-Nezhad, 2008). Their approach is built on the fact that to find desirable consistent clauses in ILP systems it is necessary to evaluate a large number of inconsistent clauses, and such consistent clauses are located at the fringe of the search space. The approach is composed of two components. The first one, called *Quick Generalisation* (QG), carries out a random-restart stochastic bottom-up search to efficiently generates a population of consistent clause on the fringe of the refinement graph search without needing to explore the graph in detail. The second component is a Genetic Algorithm which evolves and re-combines those seeded clauses, instead of performing the A* of Progol

system. The experiments performed in that work indicate that QG/GA algorithm can be more efficient than the standard refinement graph search of Progol system, while generating similar or better solutions.

The approaches just reviewed were both framed in a single clause search algorithm. One consequence of this is that the statistically assessed performance ranking of individual strategies may not be representative of their performance when used for an incremental entire-theory construction due to the statistical dependence between the successive clause search procedures.

Another issue concerns the time spent to evaluate a candidate hypothesis. Even though the search space is reduced because of SLS methods, there is a large amount of time used to check whether a hypothesis should be chosen as the next candidate solution. In the next section we present methods to learn theories which try to overcome both covering and hypothesis evaluation pitfalls.

5.3.2 Stochastic Local Search in ILP for Searching the Space of Theories and/or using Propositionalization

The standard greedy covering algorithm employed by most ILP systems is a shortcoming of typical ILP search. There is no guarantee that greedy covering will yield the globally optimal hypothesis; consequently, greedy covering often gives rise to problems such as unnecessarily long hypothesis with too many clauses. To overcome the limitations of greedy covering, the search can be performed in the space of entire theories rather than clauses (Bratko, 1999). However, there is a strong argument against this: the search space composed of theories is much larger than the search space of individual clauses. It is interesting then to apply an efficient search strategy such as SLS for searching hypothesis in the space of theories and hence uniting the benefit of both techniques.

Stochastic local search algorithms for propositional satisfiability benefit from the ability to quickly test whether a truth assignment satisfies a formula. As a result, many possible solutions (assignments) can be tested and scored in a short time. In contrast, the analogous test within ILP—testing whether a hypothesis covers an example—takes much longer, so that far fewer possible solutions can be tested in

the same time.

Thus, considering both motivations above, in a recent work (Paes et al., 2006b) we have applied stochastic local search to ILP, but not to the usual space of first-order Horn clauses. Instead, we used a propositionalization approach that transforms the ILP task into an attribute-value learning task. In this alternative search space, we can take advantage of fast testing as in propositional satisfiability. Additionally, we use a non-covering approach to search in the space of theories since now we are dealing with a more efficient search strategy and we also turn the relational domains into a simpler propositional one. Then, we use a SLS algorithm to induce k-term DNF formulae, which performs refinements on an entire hypothesis rather than a single rule (Rückert and Kramer, 2003).

Propositionalization

Propositionalization is a method to compile a relational learning problem to an attribute-value problem, which one can solve using propositional learners (Lavrac and Dzeroski, 2001; Krogel et al., 2003). During propositionalization *features* are constructed from the background knowledge and structural properties of individuals. Each feature is defined as a clause in the form $f_i(X) := Lit_{i,1}, ..., Lit_{i,n}$ where the literals in the body are derived from the background knowledge, and the argument in the clause's head is an identifier of the example. The features are the attributes which form the basis for columns in single-table (propositional) representations of the data. If such a clause defining a feature is called for a particular individual and this call succeeds, the feature is set to "true" in the corresponding value column of the given example; otherwise it is set to "false".

There are several propositionalization systems such as RSD (Železný and Lavrac, 2006) and SINUS (Lavrac and Dzeroski, 1994), among others. In this work we used RSD as the base propositionalization system. RSD constructs features by discovering statistically interesting relational subgroups in a population of individuals[1].

[1] RSD is publicly available at http://labe.felk.cvut.cz// zelezny/rsd

Stochastic Local Search in k-term DNF Relational Propositionalised Domain Learning

After propositionalising the relational domain, we apply an SLS algorithm to learn k-term DNF formulae from the feature-value table. The aim in k-term DNF learning is to induce a formula of k terms in disjunctive normal form, where each term is a conjunction of literals (Kearns and Vazirani, 1994). k-term DNF learning is a NP-hard problem of combinatorial search. The SLS algorithm designed in (Rückert and Kramer, 2003) to solve k-term DNF learning is reproduced here in Algorithm 5.6.

The algorithm starts generating randomly a hypothesis, i.e., a DNF formula with k-terms and then refines this hypothesis in the following manner. First, it picks a misclassified example at random. If this example is a positive one the hypothesis must be generalised. To do so, a literal has to be removed from a term of the hypothesis. Now, with probability p_{g1} and p_{g2} respectively, the term and a literal in this term are chosen at random. Otherwise the term in the hypothesis which differs in the smallest number of literals from the misclassified example and the literal whose removal from the term decreases the score at most are chosen. On the other hand, if the example is a negative one, it means that the hypothesis must be specified. Therefore, a literal has to be added in a term. The term is chosen at random from those ones which cover the misclassified negative example. In a similar way to the last case, either with probability p_s the literal to be added in this term is chosen at random or a random literal which decreases the score at most is taken. This iterative process continues until the score is equal to zero or the algorithm reach a maximum number of modifications. All the procedure is repeated a pre-specified number of times.

It is important to mention that Algorithm 5.6 performs refinements of an entire hypothesis rather than a single rule. A detailed analysis of SLS performance compared to WalkSAT shows the advantages of using SLS to learn a hypothesis as short as possible (Rückert and Kramer, 2003).

Experiments and remarks about them Two ILP benchmarks were considered in the paper (Paes et al., 2006b): the East-West Trains (Michalski and Larson, 1977)

Algorithm 5.6 A SLS algorithm to learn k-term DNF formulae (Rückert and Kramer, 2003)

Input: Integers k and $maxSteps$; probability parameters p_{g1}, p_{g2} and p_s; a set of examples E in attribute-value form
Output: A k-term DNF formula

1: $H \leftarrow$ a random generated DNF formula with k terms
2: $steps \leftarrow 0$
3: **while** $score(H) \neq 0$ and $steps < maxSteps$ **do**
4: $\quad steps++$
5: $\quad ex \leftarrow$ a incorrectly classified example under H, get at random from E
6: $\quad$ **if** ex is a positive example **then**
7: $\quad\quad$ **with** probability p_{g1} **do**
8: $\quad\quad\quad t \leftarrow$ a random term from H
9: $\quad\quad$ **otherwise**
10: $\quad\quad\quad t \leftarrow$ the term in H that differs in the smallest number of literals from ex
11: $\quad\quad$ **with** probability p_{g2} **do**
12: $\quad\quad\quad l \leftarrow$ a random literal in t
13: $\quad\quad$ **otherwise**
14: $\quad\quad\quad l \leftarrow$ the literal in t whose removal decreases $score_L(H)$ most;
15: $\quad\quad H \leftarrow H$ with l removed from t
16: $\quad$ **else if** ex is a negative example **then**
17: $\quad\quad t \leftarrow$ a (random) term in H that covers ex;
18: $\quad\quad$ **with** probability p_S **do**
19: $\quad\quad\quad l \leftarrow$ a random literal m so that $t \wedge m$ does not cover ex;
20: $\quad\quad$ **otherwise**
21: $\quad\quad\quad l \leftarrow$ a literal whose addition to t decreases $score_L(H)$ most
22: $\quad\quad H \leftarrow H$ with l added to t

and Mutagenesis Data (Srinivasan et al., 1996). They were both propositionalised by RSD, producing a set of features in attribute-value form. Then, the K-term DNF SLS learner were compared to Aleph in its default mode and to Aleph using GSAT search as explained in section 5.3, both using the original relational dataset. Additionally, the propositionalised domains were given as input to Part (Frank and Witten, 1998) and Ripper (Cohen, 1995), two popular rule learners algorithms. Besides the fact that they do not use SLS, they also differ from K-term DNF learner in the use of a covering approach. The results were compared in terms of compression achieved, cpu time consumed and predictive accuracy. They indicated that DNF/SLS performs faster w.r.t. all other tested methods when it comes to short theories (in number of rules). Comparing to relational methods, the performance

gap was significantly large (in orders of magnitude), while corresponding predictive accuracy does not favor either of SLS/DNF or the relational methods. Comparing to the propositional methods, this performance gap is much smaller, while SLS/DNF's short theories exhibit slight superiority in terms of predictive accuracy.

There are other approaches to induce hypothesis using SLS with or without propositionalization. For instance, (Serrurier and Prade, 2008) employs Simulated Annealing to induce hypothesis directly - do not using neither propositionalization nor covering. The candidate hypothesis are generated by a neighbourhood relationship derived from a refinement operator defined over hypothesis. In this case, the neighbourhood of a current hypothesis H is composed by adding or removing clauses from H or still by applying a refinement operator (downward or upward) on each one of its clauses. In (Joshi et al., 2008; Specia et al., 2009) was developed an approach to construct features randomly in order to build modes to assist the task of Word Sense Disambiguation. A randomised search procedure based on GSAT and using theory-error-guided sampling is designed for dynamically construct the features and generate the output model. The procedure is composed of two nested steps: the outer loop iterates R times, where R is a pre-defined number of restarts and the inner loop executes M times, where M is a pre-defined number of local moves. In the outer loop, a sample of n acceptable features is generated, where a feature is considered as acceptable it covers at least s examples, has a minimal pre-defined precision p and obeys constraints of the language. A model is then constructed from this set of features. The inner loop iteratively selects a new subset of features based on the errors made by the current features on the current model.

Chapter 6

Revising First-Order Logic Theories through Stochastic Local Search

6.1 Introduction

Usually, a First-Order Theory Revision system perform search in three steps. First, it searches for points in the theory responsible for misclassifying an example. Second, it searches for possible modifications to be implemented within each revision operator, including addition and deletion of antecedents in the body of clauses. And finally, it searches from a number of available revision for the one which will be responsible for implementing a modification to the theory. In each one of these searches, a system such as FORTE and its descendants systems follow an enumerative strategy, engendering large search spaces that may grow to be intractable, according to the factors below.

1. The number of misclassified examples, since the revision system traverses each example's proof looking for faulty points;

2. The size of the initial theory, since every clause on it might be potential revision points;

3. The number of clauses responsible for misclassifying an example, since the revision system proposes modifications to each one of them;

4. The size of the knowledge base and background knowledge, since antecedents must have to be generated and added to the body of clauses, as possible modifications to be implemented on the theory.

Additionally, theory revisions systems do tackle whole theories instead of step-wise search for individual clauses as most ILP systems do. Search over whole theories is known to be a hard problem (Bratko, 1999). As a result, traditional theory revision systems must search over extremely large spaces, and can become rather inefficient and even intractable.

The last years have shown that stochastic local methods, originally designed to solve difficult combinatorial propositional problems (Selman et al., 1992; Selman et al., 1996; Rückert and Kramer, 2003), can also perform well in a variety of applications. Moreover, combining Stochastic Local Search with inductive Logic Programming has shown very substantial improvements in efficiency, with little or no cost in accuracy (Srinivasan, 2000; Paes et al., 2006b; Železný et al., 2006; Muggleton and Tamaddoni-Nezhad, 2008). Such results motivate the contribution of this chapter. Further, we take Trefheten's Maxim No. 30 into high consideration, which states that if the search space is huge, the only reasonable way to explore it is at random (Trefethen, 1998). Thus, we aim to achieve a balance between efficiency and efficacy, by decreasing the negative impact of the mentioned factors on the running time of theory revision. To do so, we sacrifice completeness in favor of finding good solutions rather than optimal ones, by means of SLS techniques. Stochastic components are included in the key searches of the revision process, namely:

1. Search for revision points: A random decision may return a subset of the revision points instead always returning all of them.

2. Search for literals: as the proposals of modifications are dominated by addition and deletion of antecedents, one may benefit from randomising antecedent search (Paes et al., 2007a; Paes et al., 2007b).

3. Revision search: rather than proposing all revisions, one might enumerate the possible modifications and choose one to implement at random (Paes et al.,

2007b).

Preliminary experiments with the theory revision system FORTE showed good promise from introducing stochastic search at the last two searches above (Paes et al., 2007a; Paes et al., 2007b). This chapter revisits that work enhancing it by designing a number of stochastic search in every step of YAVFORTE system. Note that in (Paes et al., 2007a; Paes et al., 2007b) stochastic components were introduced in original FORTE system and in the present work they are built upon YAVFORTE system, including the bottom clause to bound the search space and mode declarations as the language.

The outline of this chapter is as follows. Firstly, section 6.2 brings the stochastic algorithm developed to search for the revision points. Next, stochastic algorithms applied within the revision operators, for choosing literals to be added to or removed from a clause, are devised in section 6.3. Then, we present a number of SLS algorithms for deciding which revision operator will be responsible for modifying the theory in section 6.4. Finally, experimental results are presented in section 6.5, followed by conclusions in section 6.6.

6.2 Stochastic Local Search for Revision Points

FORTE-based systems follow the key steps below in order to generate revision points:

1. Identify the misclassified instances;

2. Through the misclassified examples, find the clauses and/or antecedents responsible for such misclassifications. These points will compose the set of revision points;

3. Calculate the potential of each revision point. Remember that the potential is the number of examples which identify the necessity of modifying the revision point;

4. Identify the relevant examples for each revision point, i.e., the examples whose provability can be affected after proposing some revision in that point. This

is essential to make the evaluation of revision operators more efficient by not proving every example at each modification, since only the proofs of relevant examples must be re-done when proposing modifications on specific points of the theory.

It can be seen from the steps above that there are two major factors working together to possibly increase the cost of searching for revision points: the set of examples and the size of the theory. This happens because each example must be tested on the theory, either to identify faulty clauses or literals or to check whether the example is relevant to the revision point. Moreover, each clause in the theory may be tested as a potential revision point. Therefore, what it is done in this thesis is to decrease the work performed in those tests by introducing a stochastic component within the search for revision points. Rather than always looking for all revision points in the theory, the stochastic component allows only a subset of that group to be sought.

The strategy developed in this chapter does not only avoid searching in the whole theory for revision points but also avoids considering all misclassified examples. It works by alternating between stochastic and complete moves according to a certain probability, a method that can be seen as an instance of Randomising Iterative Improvement and WalkSAT techniques. Thus, with a probability p_{rp} the stochastic move is taken and the procedure will look for only a subset of all the possible revision points. The size of the subset is previously defined by the user, with its default value as 1. Otherwise, a complete move is taken just as in the original algorithm.

The stochastic move works as follows to gather the subset of revision points. First, it selects a misclassified example at random. Then, revision points are collected from such an example. In case this single misclassified instance already produces the required number of revision points, the procedure stops. If it produces more than that, the number is chosen at random from them. If the instance does not have enough revision points, the procedure proceeds to collect more revision points by choosing another misclassified example at random. After collecting the subset of random revision points, it is time to find out the relevant examples. This

is necessary to compute the potential and also to consider only those examples to be proved again when evaluating a modification on the revision point. Note that in case the probability p_{rp} is 100% and the subset is required to be composed of only one revision point, the approach employed here follows the same line of thought of (Rückert and Kramer, 2003), although there they address propositional learning. Algorithm 6.1 brings the procedure for collecting revision points using a stochastic component.

Algorithm 6.1 SLS Algorithm for generating revision points

Input: A set of positive and negative examples E, divided into ECC, the correctly classified examples and EIC, the incorrectly classified examples, Theory T, probability p_1, A integer k
Output: A set of revision points RP

```
1:  RP ← ∅
2:  with probability p1 do
3:      while #RP < k do
4:          ex ← a misclassified instance chosen at random from EIC;
5:          num_rp = k − #RP
6:          RP_ex ← at most num_rp revision points generated from ex;
7:          RP ← RP ∪ RP_ex;
8:  otherwise
9:      RP ← the revision points generated from all misclassified examples in E
10: for each revision point rp ∈ RP do
11:     identify the relevant examples for RP;
12:     calculate the potential of RP;
13:     sort RP by potential;
14: return rps;
```

It is important to stress that examples from both classes have the same chance to be chosen at random. This is particularly important when we have skewed datasets, with one class largely dominating the other on the number of examples. Algorithm 6.1 introduces the stochastic component as an alternative path in Algorithms 2.2 and 2.3, by avoiding in stochastic moves to search for revision points at each clause of the theory and not considering all misclassified examples. In this way, roughly speaking, instead of getting a complexity time bounded by the number of training examples times the size of the theory, we get for stochastic moves a time complexity limited to the number of required revision points, which in the best case is only 1. Note that in the complete case, the theory is traversed even

to the correctly classified examples, because it is in this moment that the relevant examples are collected. Although the stochastic search still has to find the relevant examples for the revision points in the set of all training examples, this is also a reduced search, since it is restricted to the set of revision points instead of traversing the whole theory. Additionally, in case the search returns only one kind of revision points, two distinct cases be addressed: (1) the revision point is a generalisation one and then only unprovable examples are considered (true negative and false negative) or (2) the revision point is a specialisation one and only provable examples (true positive and false positive) are considered.

6.3 Stochastic Local Search for Literals

The main goal of introducing antecedents to a clause is to stop proving negative examples while continues covering as much of the originally proved positive examples as possible. To do so, the operator can proceed from two approaches. Either it uses a hill-climbing procedure, where at each iteration the antecedent which improves the score at most is chosen to be added in the clause or it uses the relational pathfinding algorithm, where more than one antecedent can be added to a clause at once. These two approaches can also be combined, with the relational pathfinding algorithm being executed and, next, antecedents being added to a clause through the hill-climbing algorithm. Both approaches consider the bottom clause generated from a covered positive example as their search space.

Similarly, the delete antecedents operator has the goal to make the clause to start proving positive examples while still does not proving as much of the negative examples as possible. To achieve its goal, this operator either removes one antecedent at once from the clause, using a hill-climbing approach, or it can delete multiple antecedents at once to escape from maxima local. However, this last approach is only used when the latter does not produce any results, since it is expensive to list and test the combination of all possible literals to be removed from the clause. Both approaches require the modes language to be obeyed after a removal.

Add antecedentes and delete antecedents are the basic operations performed into all operators of YAVFORTE with the exception of delete rule: Specialisation

is performed either by adding antecedents to a clause or deleting clauses from the theory; generalisation is achieved by either deleting antecedents from a clause or creating new clause. In this last case it is possible to create a clause from another one, by leaving the original one in the theory and deleting antecedents followed by adding antecedents from its copy. It is also possible to create a completely new using the add antecedents operator in a clause with a predicate in its head, properly defined in a modeh declaration.

Regarding addition of antecedents there are three main factors impacting the search space of the bottom clause: (1) the size of the intentional and extensional background knowledge, (2) the amount of different modeb definitions to the predicates together with the recall of each one of them, and (3) the value set to variables depth. In fact, as it was shown in Algorithm 5.3, the cardinality of a bottom clause is bounded by $r(|M|j + j-)^{ij^+}$, where $|M|$ is the cardinality of the set of modes declarations, j^+ is the number of + type occurrences in each modeb in M plus the number of – type occurrences in each modeh, j^- is the number of – type occurrences in each modeb in M plus the number of + type occurrences in each modeh, r is the recall of each mode $m \in M$, and i is the maximum variable depth. Thus, the bottom clause generates a search space of exponential size w.r.t. the maximum variable depth. In case the recall r is defined as $*$, which is the most common case, all the possible instantiations of a literal are going to be collected in the BK. Because of that, the size of the BK also influences the cardinality of the bottom clause.

Possibly, each element of the bottom clause may be tested on each example relevant to the clause being specialised, excluding those who does not have variables compatible to the mode declarations of the clause (although they also slightly influence the running time since they must be tested to check the compatibility). Additionally, in the worst case, to specialize one single clause it is necessary to pick up literals from the bottom clause as many times as the maximum size set to a clause. Considering all these factors, addition of antecedents is an expensive operation and still performed many times during the whole revision process. Not counting the relational pathfinding algorithm which is expensive for itself (Richards and Mooney, 1992). Therefore, we intend to make the add antecedents operator,

and consequently the revision process, more efficient by introducing stochastic components on this. Once again, we sacrifice completeness to gain efficiency when proposing modifications on the theory by adding antecedents to clauses or creating new rules.

Deleting antecedents is obviously a less expensive operation than adding antecedents. The search space is composed of only the literals in the clause. Because of that at each iteration an antecedent is deleted the search space of the next iteration is reduced. Thus, in the worst case the search space of tested literals at each iteration has all antecedents in the body of the current clause. Note that sometimes the search space is is less than the size of the clause because some literals can make the clause incompatible to the modes definitions when removed from it. However, they still increases the running time, although slightly, since it is necessary to check if they can or cannot be a candidate to be deleted. Additionally, in the worst case, the operation of deleting antecedents may be performed $|clause|$ times in case deletions always improve the score. Thus, although benefiting less than when adding antecedents, we can also improve running time of the proposals of modifications by making the delete antecedent operator more efficient. Aiming this goal, we also introduce stochastic search when deleting antecedents, either when proposing generalisations on a single clause or when generalising the theory by creating a new rule from an existing one.

Stochastic versions of the delete antecedents and add antecedents algorithms were developed, performing according to the following strategy. They may perform either a random or a greedy move, depending upon a fixed probability. While the greedy move is the same for both approaches, since in this case the original algorithm is maintained, the random move differs for each approach. Next we devise each approach separately.

6.3.1 Stochastic Component for Searching Literals

The algorithm introducing a stochastic component follows a stochastic hill climbing technique and adopts a conservative strategy even when performing a random move, by maintaining the requirement of improving the value of the evaluation function,

and by refining clauses in the same way as it is done in the original algorithm. The stochastic component is employed for choosing the next candidate clause. The decision of not allowing bad moves to escape from local maxima is arguably justified because adding/deleting a sequence of literals at once, using the relational pathfinding algorithm or the algorithm for deleting multiple antecedents, may already be able of escaping from local maxima. Basically, a random walk is carried out by taking a random step for choosing the next candidate clause with a fixed probability p_l, In case the probability p_l is not achieved, the original greedy hill climbing algorithm is performed. This algorithm is built upon the following decisions.

- **Defining the search space:** Candidate clauses are formed by adding/deleting a single literal or a sequence of literals in the current clause, depending on the algorithm employed (greedy hill climbing or relational pathfinding/deletion of multiple antecedents). The literals to be added/deleted are elected differently, depending upon the move is greedy or stochastic. In case the move is greedy, the approach is pure hill climbing in the sense the current clause and candidate clauses differ by only one literal and the goal is to add antecedents, the set of possible antecedents to be added to a clause is composed by literals taken from the bottom clause. Similarly, when adding a sequence of literals, the paths are created considering the literals of the bottom clause. When deleting antecedents in the greedy move, candidate clauses are created either by removing a single literal or by removing a combination of literals. In case the move is random and the goal is to add single antecedents, a literal picked at random from the bottom clause is added to the current clause to form a candidate clause. Similarly, a path created from the literals of the bottom clause in relational pathfinding algorithm is chosen at random. To delete antecedents from a clause in a stochastic move, the body of the clause is randomised and then a literal is chosen. In a similar way, combinations of literals are randomised and one of them is taken at random. In all of the cases, the candidate clause must be valid according to the modes declarations.

- **Choosing the next clause:** Since this conservative approach requires the next clause improves the current score, in all cases listed above this demand must be attended. However, in greedy moves, all candidate clauses are tested on the examples in order to calculate their score and the one improving the score at most is the chosen one, while in random moves, the first randomly generated candidate clause improving the score is chosen. In others words, to choose at random the next clause, a single candidate clause formed as explained in the last topic has its score computed using the set of examples. If this clause already improves the score, it is going to be the next clause. Otherwise, it is necessary to choose another candidate clause, which is clearly formed as explained above, by selecting a random literal or a sequence of literals. This procedure continues until finding a candidate clause improving the score or to find out that there is no clause able to do that, finishing the operator procedure. The only exception is the relational pathfinding algorithm, since its procedure allows a path to be chosen if the score is not changed (neither increased nor decreased). It only allows that because after it runs, hill climbing is employed to further specialize the resulting clause. Note that in the best case only one candidate clause is evaluated, which makes the run time of the procedure independent on the size of the bottom clause in case of specialisations, or on the current clause, in case of generalisations. In the worst case all possible candidate clauses are evaluated as in greedy steps. Choosing next clause in random moves greatly benefits from do not computing score for each possible candidate, which is the most expensive task performed inside the original algorithms since it is necessary to check the provability of each relevant example.

- **Stop criteria:** As usual in hill climbing approaches, the algorithm stops when there are no more candidate clauses improving the score, either because the set of generated candidate clauses are not able to do that, or because there are no further valid clause to be evaluated.

Algorithm 6.2 substitutes hill-climbing addition of antecedents, exhibited as Algorithm 3.3 in both add-antecedent specialisation operator and second phase of the add-rule generalisation operator. The algorithm starts by generating the bottom clause from a covered positive example, as it is done in the original Algorithm. After that, it performs a random walk, following the approach of algorithms such as WalkSAT, and decides the type of the move, based on a fixed probability p_{ls}. In case p_{ls} is not reached, the algorithm performs a greedy hill climbing step, exactly as it done in the original algorithm: all valid (according to modes) candidate clauses formed by adding the literals from the bottom clause to the current clause are evaluated on the examples. Then, the candidate clause improving the score at most is selected. If there is no such improving clause, the procedure returns nothing. If the probability p_{ls} is reached, a random step is taken: a literal is selected at random from the bottom clause and added to the current clause. After the candidate clause is validated relative to the modes, it is evaluated using the examples. In case such a clause improves the current score, it is chosen to replace the current clause. Otherwise, it is discarded and another candidate clause is selected. This procedure continues until finds a clause improving the score or until exhausting all the possibilities. Finally, the candidate clause replaces the current clause (if there is one) and the algorithm proceeds to the next iteration. This procedure is performed until there is no further clause improving the score or if is reaches the maximum size defined to clauses.

Relational pathfinding algorithm provides a sequence of antecedents to be introduced in a clause. The algorithm searches for all possible sequences and chooses the one with the highest score. In case of a tie, the smallest sequence is chosen. A stochastic version of this algorithm selects the sequence to be added to the current clause according to a stochastic decision: with a probability p_{ls} it chooses a sequence at random from all the possible generated paths; otherwise it proceeds as in the original algorithm. We do not generate paths at random, since this algorithm tries to find a sequence of literals connecting the variables in the head of the clause, and to introduce a randomness component into this process could either disregard a possible valid sequence or to force the procedure to backtrack to several previ-

Algorithm 6.2 Algorithm for adding antecedents using hill-climbing SLS

Input: A clause C, CL, maximum size of a clause, p_{ls}, the probability for deciding which move is going to be taken
Output: A (specialised) clause C'

1: **repeat**
2: $currentScore \leftarrow$ compute score of C;
3: $BC \leftarrow$ createBottomClause(...);
4: **with** probability p_{ls} **do**
5: **repeat**
6: $ante \leftarrow$ an antecedent chosen at random from BC, whose input variables are already in C (therefore it obeys modes);
7: $C' \leftarrow C$ with $ante$ added to it;
8: $candidateScore$ score of C';
9: **if** $candidateScore > currentScore$ **then**
10: $C \leftarrow C'$
11: $currentScore \leftarrow candidateScore$
12: **else**
13: $BC \leftarrow BC - ante$
14: **until** $C = C'$ or $BC \neq \emptyset$
15: **otherwise**
16: **for each** antecedent $ante \in BC$ **do**
17: $C' \leftarrow C$ with $ante$ added to, in case $C + ante$ obeys the modes declarations;
18: $candidateScore$ score of C';
19: $bestClause \leftarrow$ candidate clause with the highest $candidateScore$
20: **if** $candidateScore > currentScore$ **then**
21: $C \leftarrow bestClause$
22: $currentScore \leftarrow candidateScore$
23: remove $ante$ from BC
24: $FPC \leftarrow FPC-$instances in FPC not proved by C;
25: **until** $FPC = \emptyset$ or there are no more antecedents in BC or it is not possible to improve the score of the current clause or $|C| = CL$
26: **return** C

ous points. As it was said before, this algorithm is quite expensive by itself and therefore introduce more backtracks on it goes contrary to our primary objective of reducing run time. Therefore, the benefit the stochastic algorithm brings is to avoid computing score considering the set of examples for each possible sequence, which is obviously an expensive task. Algorithm 6.3 shows such procedure.

Algorithm 6.3 Stochastic Relational-pathfinding

1: generate all possible sequence of antecedents through relational_pathfinding algorithm and the Bottom clause;
2: **with** probability p_{l2} **do**
3: choose a sequence at random;
4: **otherwise**
5: choose a sequence with the highest score or the one with less antecedents in case of a tie;

The delete-antecedent operator benefits less from stochastic local search than add-antecedent, since the search space is restricted to goals in the clause, and is therefore much smaller. The hill-climbing stochastic algorithm for antecedent deletion is shown in Algorithm 6.4. Notice that delete-antecedent is also part of add-rule, with the latter using it in its first phase. The algorithm follows exactly the same random walk approach as previous algorithms seen in this section. First, it decides which type of move it is going to take, namely, a greedy move or a random move, based on a fixed probability p_{lg}. In case the move is greedy, it uses the original algorithm to propose deletions of antecedents. Otherwise, it selects at random a literal to be removed from the clause. Both cases require an improvement on the score to indeed remove a literal from the clause.

A stochastic version of the delete multiple antecedents algorithm was also developed. However, this algorithm is executed only when the hill-climbing approach for deleting antecedents is not capable of deleting any antecedent. It is quite simple and it only differs from the original one by the stochastic decision introduced to decide which movement to make, greedy or random. A greedy move previews all possible combinations of literals from the body of the clause to choose the best among them, while a random move chooses the first combination improving the current score. Note that both cases only delete antecedents if after the deletion the

Algorithm 6.4 Algorithm for deleting antecedents using hill-climbing SLS

Input: A clause C, p_{lg}, the probability for deciding which move is going to be taken
Output: A (generalised) clause C

1: **repeat**
2: $currentScore \leftarrow$ compute score of C;
3: $antes \leftarrow$ antecedents from the body of C;
4: **with** probability p_{lg} **do**
5: **repeat**
6: $ante \leftarrow$ an antecedent chosen at random from $antes$, whose removal from C still makes it valid relative to modes;
7: $C' \leftarrow C$ with $ante$ deleted from it;
8: $candidateScore \leftarrow$ compute score of C';
9: **if** $candidateScore > currentScore$ **then**
10: $C \leftarrow C'$
11: $currentScore \leftarrow candidateScore$
12: **else**
13: $antes \leftarrow antes - ante$
14: **until** $C = C'$ or $antes = \emptyset$
15: **otherwise**
16: **for each** antecedent $ante \in antes$ **do**
17: $C' \leftarrow C$ with $ante$ deleted from;
18: $candidateScore \leftarrow$ compute score of C';
19: $bestClause \leftarrow$ candidate clause with the highest $candidateScore$
20: **if** $candidateScore > currentScore$ **then**
21: $C \leftarrow C'$
22: $currentScore \leftarrow candidateScore$
23: **until** no antecedent can improve the score;
24: **return** C

clause continues to obey modes declarations. This algorithm is better visualised in 6.5.

Algorithm 6.5 Stochastic version of the algorithm for deleting multiple antecedents

1: **with** probability p_{lg} **do**
2: generate all possible sequence of antecedents from the clause;
3: choose a sequence at random;
4: **otherwise**
5: choose a sequence with the highest score;

6.4 Stochastic Local Search for Revisions

YAVFORTE employs two specialization revision points, namely delete rule and add antecedent, and three generalisation revision points, namely delete antecedent, add rule (creating a rule by an existing one, by copying the original, deleting antecedents from it, followed by adding antecedents on it) and add new rule (creating a clause from scratch). Even employing all those revision operators, it may be the case that the set of revision points has only few components, either because the theory has only few failure points or because the stochastic search for revision points was applied and only a subset of the revision points is given back. In such a situation, the search space for selecting the operator which will be indeed responsible for modifying the current theory can be small enough to not largely influence the runtime of the revision process. However, this small search space does not always occur, making the choice of the revision operator be an important factor of cost during the revision process, since it is necessary to evaluate each possible revision yielded by the proposal of each revision operator on each matching revision point.

In this way, the revision process can also benefit from stochastic search to reduce runtime when selecting the revision operator which will be the responsible for implementing some modification on the theory. Additionally, the revision can also take advantage of stochastic local search techniques to escape from local maxima. Aiming that goals, we designed four stochastic version of the top level algorithm of YAVFORTE. They differ mainly in the decision made to implement a revision, since in some algorithms a bad move can be executed. The algorithms are called as follows.

1. Stochastic greedy search with random walk
2. Stochastic hill-climbing search with random walk
3. Stochastic hill-climbing with stochastic escape
4. Simulated annealing search

Next we discuss each one of these algorithms.

6.4.1 Stochastic Greedy Search for Revisions with Random Walk

This approach follows Randomised Iterative Improvement technique and its most famous instance, namely WalkSAT algorithm, and accordingly, it performs random walks, alternating between greedy and stochastic moves, based on a fixed probability. In case this fixed probability is not reached, the algorithm performs a greedy move, by proposing all possible modifications on the theory though the application of each matching revision operator to each found revision point, in the same fashion the original algorithm does. Then, the best proposed revision is chosen to be implemented. Note that this move is greedy only, rather than greedy hill climbing as in original algorithm and therefore the score may be worse than the current one.

In case the fixed probability is reached, the algorithm performs a stochastic move, selecting at random a revision to be implemented from all the possible ones. There is no requirement on the selected revision: it is neither mandatorily the best possible revision nor it is required to improve current score. It is just a revision proposed on a revision point by a matching revision operator. Because of that, it is not necessary to explicitly propose and evaluate all the possible revisions, since no assumptions are made on the score. Thus, it is enough to *enumerate* the possible revisions through the revision points and the revision operators suitable to be applied on each of those revision points. For each revision point, the list of possible revisions gets an entry containing a tuple with the revision point plus a matching revision operator. For instance, regarding a single specialisation revision point SRP and assuming both revision operators are employed in the revision process, there is

going to be two tuples in the list: one containing SRP plus add-antecedent and another containing SRP plus delete-rule. The same is done for generalisation revision points. From such enumeration, a revision can be chosen at random and then be implemented.

Note that bad moves are always allowed in this greedy stochastic approach. On one hand, this ability makes the algorithm capable of always escaping from local maxima. On the other hand, some moves can conduct the theory to such a damaged state that the revision process will not be able to recover from it. However, modifications are performed on failure points and revision operators are designed to propose worthwhile revisions on those points, which makes the risk of really deteriorating the theory very small. To summarise, the key ideas of Stochastic Greedy Search with Random Walk algorithm are:

- **Composing the space of candidate hypothesis:** the search space is formed differently depending on the type of the move. If the move is greedy, the search space is composed of all possible proposals of revisions applied on each revision point by the matching revision operators. If the move is stochastic, the candidate hypothesis are also all the possible revisions, however, as one of them are going to be chosen at random, the candidates are not in fact proposals of revisions, but instead a representative tuple of the real revision. Each tuple contains the revision point and a possible revision operator to be applied on it.

- **Choosing the revision to be implemented:** In a greedy move, all revisions composing the search space of candidate hypothesis are evaluated by an evaluation function and the best revision is indeed implemented on the current theory. In a stochastic move, a tuple representative of one real revision is chosen at random to be implemented. As the revision has not been already proposed before to be chosen as it happens in greedy moves, it is necessary to check if it is really possible to modify the theory with that revision. In other words, it may be the case the chosen revision cannot produce any modification in the theory. This is the case, for example, of trying to delete a rule and it

is the last one for a top level predicate, or still if is not possible to delete or add antecedents on the clause. In such a situation, the procedure must choose another revision from the list until the list is empty. If it is possible to propose a revision from the tuple, then it is implemented on the theory.

- **Stop criteria:** The revision process stops under three circumstances: (1) when there are no more revisions to be implemented, (2) when the revision reaches a maximum score on the training set (for example, in case the evaluation function is accuracy, the maximum score could be 1.0) or (3) when the procedure implements a maximum number of revision (performs a maximum number of steps).

The procedure is exhibited in Algorithm 6.6 as a simplified and replacing version of Algorithm 3.1. By simplified version we mean some lines are omitted, specially those concerning the application of only required revision operators. The algorithm starts by computing the score of the current theory, since one of the stop criteria is the score reaches a maximum value. Next, it starts a loop which is going to stop according to the criteria established just above. Inside the loop, the first matter to worry about is finding revision points. Note that either the original algorithm or the stochastic algorithm may be used to this task, depending on the user preference. Then, as usual, the move must be chosen. In a stochastic move, it is necessary to find a random "implementable" revision. To do so, the revisions are enumerated considering each revision point and revision operators that are applicable on them. A revision is then selected at random and the system tries to implement in on the current theory. If it is possible, the procedure proceeds to the next iteration. On the contrary, another revision is chosen at random, until one of them can indeed be implemented or there are no more possible revisions. In case the move is greedy, proposals of revisions are generated and scored in the same way the YAVFORTE top-level algorithm does. However, here the best revision is implemented even though the score is not improved.

While Stochastic Greedy strategy is able to escape from local maxima, it is also a risky approach, since as we discussed before there is a chance of the theory de-

Algorithm 6.6 Stochastic Greedy Search for Revisions with Random Walk

Input: An initial theory T, A Background Knowledge FDT, a set of examples E, integer $maxSteps$, real $maxScore$
Output: A revised theory T'

1: $score \leftarrow$ score of T
2: $steps \leftarrow 0$
3: **repeat**
4: generate revision points
5: **with** probability p_{rev} **do**
6: $possibleRevisions$ possible revisions enumerated from the revision points and respective revision operators
7: **repeat**
8: $nextRevision \leftarrow$ a revision chosen at random from $possibleRevisions$
9: $T' \leftarrow$ implements $nextRevision$
10: **until** $T \neq T'$ or $possibleRevisions = \emptyset$
11: **otherwise**
12: generate all possible revisions from the revision points and respective revision operators
13: compute score of each proposed revision
14: $nextRevision \leftarrow$ revision with the highest score
15: $T \leftarrow$ implements $nextRevision$
16: $score \leftarrow$ score of T
17: steps ++;
18: **until** $score >= maxScore$ or $steps = maxSteps$ or T has not been modified

teriorates in one iteration and never recover again. Next section we present a second algorithm for selecting a revision to be implemented that is based on a Stochastic Hill-Climbing strategy, and therefore does always request a revision improving the current score.

6.4.2 Stochastic Hill-Climbing Search for Revisions with Random Walks

The Stochastic Hill Search with Random Walks algorithm, as the previous algorithm, also alternates between greedy and stochastic moves. However, However, whatever the move is, it is required an improvement in the score to go ahead. In this way, it is the most similar to the original algorithm comparing to all the approaches designed in this section. As usual, with a fixed probability p_{rev} the algorithm chooses at random a revision to be implemented from the list of possible revisions. Such a list is an enumeration of *revision point – operator* tuples formed exactly in the same way that the previous discussed algorithm.

In order to guarantee the improvement in the score the revision chosen at random is evaluated according to some function and it is verified if its score is better than the current one. If so, the revision is implemented. If do not, another revision is chosen at random until the procedure finds a revision with a score better than the current one, or there are no more possible revisions to be implemented. The benefit in runtime brought by this approach relies on the fact that in random moves it is not necessary to explicitly propose and compute the score of all possible revisions. Proposing and computing all revisions in random moves will only happen in very unlikely cases where there is only one revision improving the score and it is the last to be chosen.

When the fixed probability p_{rev} is not reached, the move is greedy. As the approach is greedy hill climbing, it is necessary to propose and evaluate all the possible modifications on the theory and then to choose the best one, just as the original algorithm does. The revision process stops when there are no more revisions capable of improving the score. The key components of the strategy are as follows.

- **Composing the space of candidate hypothesis:** The set of candidate

hypothesis is composed of the same elements as in the previous stochastic algorithm.

- **Choosing the revision to be implemented:** In a greedy move, all revisions composing the search space of candidate hypothesis are evaluated and the best revision is indeed implemented on the current theory, only if it improves the current score. In a stochastic move, a tuple representative of one real revision is chosen at random to be implemented. As the revision has not been already proposed before to be chosen as it happens in greedy moves, it is necessary to check if it is really possible to modify the theory with that revision and also if it is able of improving the current score. In case a revision chosen at random is implementable and improves the score, it is indeed implemented. Otherwise, another revision must be chosen from the list of possible revision, until the list is empty.

- **Stop criteria:** The revision process stops under two circumstances: (1) when there are no more revisions to be implemented so that the score is improved or (2) when the revision reaches a maximum score on the training set.

The algorithm is exhibited as Algorithm 6.7 and as before we omit some obvious lines concerning the applicability of revision operators. It starts by computing the score of the current theory, since is is the base for verifying any improvement. Then, it performs an iterative hill climbing procedure, where at each iteration the revision points are generated and either a greedy or a stochastic move is accomplished, depending on a fixed probability p_{rev}. If a stochastic move is selected, then representatives of possible revisions are collected, encompassing each revision point together with each revision operator applicable to that point. An element from that collection is chosen at random and after proposing the selected revision in a temporary variable, the new score is computed. In case there is an improvement on the score, the proposed revision is accepted as next revision. On the contrary, it is required to pick up another revision, until finding one able of improving the score or giving up because there is no such a revision. If the move is chosen to be greedy, then, as in the original algorithm, the revisions are generated by a number of

matching revision operators and scored with an evaluation function, until the maximum potential of a revision point is achieved by some revision. Afterwards, the revision with the highest score is selected and implemented in case the current score is improved with that revision. The procedure stops when the theory under revision reaches a maximum score or when there are no revision capable of modifying the current theory so that the score is improved.

Algorithm 6.7 Stochastic Hill-climbing Search for Revisions with Random Walk

Input: An initial theory T, A Background Knowledge FDT, a set of examples E, integer $maxSteps$, real $maxScore$
Output: A revised theory T'

1: $score \leftarrow$ compute score of T
2: **repeat**
3: generate revision points
4: **with** probability p_{rev} **do**
5: $possibleRevisions$ possible revisions enumerated from the revision points and respective revision operators
6: **repeat**
7: $nextRevision \leftarrow$ a revision chosen at random from $possibleRevisions$
8: $T' \leftarrow T$ after implementing $nextRevision$
9: $scoreNextRevision \leftarrow$ score of T'
10: **if** $scoreNextRevision > score$ **then**
11: $T' \leftarrow T$
12: $score \leftarrow scoreNextRevision$
13: **else**
14: $possibleRevisions \leftarrow possibleRevisions - nextRevision$
15: **until** $T' = T$ or $possibleRevisions = \emptyset$
16: **otherwise**
17: generate all possible revisions from the revision points and respective revision operators
18: compute score $scoreNextRevision$ of each proposed revision
19: $nextRevision \leftarrow$ revision with the highest score
20: **if** $scoreNextRevision > score$ **then**
21: $T \leftarrow$ implements $nextRevision$ on T
22: $score \leftarrow scoreNextRevision$
23: **until** $score = maxScore$ or T has not been modified

Next section we present an intermediate strategy between both of the stochastic algorithms discussed here. It is able to perform bad moves aiming to escape from local maxima, but only under a stochastic move.

6.4.3 Hill-Climbing Search for Revisions with Stochastic Escapes

The stochastic greedy algorithm first presented here may deteriorate the theory since it always allows bad moves to be performed. The previous algorithm, on the other hand, never allows a bad move, which can not avoid it gets stuck in local maxima. Besides, even when the move is stochastic it has to search for a revision improving the score, adding an extra factor of cost, compared to the former algorithm. Aiming to overcome those issues, a third stochastic version of the algorithm 3.1 is designed on this section. The strategy exploited here is to try escape from local maxima when performing a stochastic move and implement the best candidate otherwise. Thus, as usual, with a certain probability p_{rev}, the algorithm performs a stochastic escape, choosing a revision to be implemented at random even if its score is not better than the current one. On the other hand, when the move is greedy, this algorithm proceeds as originally, selecting the revision with the highest score and demanding such a score is better than the current one. If there is no such a revision, the algorithm continued to the next iteration in an attempt to reach the probability p_{rev} and then to perform a stochastic move. The procedure stops when it reaches a maximum number of steps or when it reaches a maximum score. As before, we summarise the key components.

- **Composing the space of candidate hypothesis:** The set of candidate hypothesis is composed of the same elements as in the previous stochastic algorithms.
- **Choosing the revision to be implemented:** In a greedy move, it behaves as the greedy component of Stochastic Hill-Climbing and hence the original revision algorithm. Thus, all revisions composing the search space of candidate hypothesis are evaluated and the best revision is implemented on the current theory, only if it improves the current score. However, in case no revision improves the score, instead of terminating like those algorithms, Hill-climbing with Stochastic Escape continues to the next iteration. In a stochastic move, it may implement a revision with a bad score, but only if it does not degradate

that much the score. To decide how much degradation it is allowed, we use a function based on a pertubation strategy (Hoos and Stützle, 2005), defined as $score + 0.5 * (Potential + score)$. Potential is the number of examples that indicated the need of revising a point and therefore is the maximum score that a revision could have.

- **Stop criteria:** The revision process stops under two conditions, namely: (1) when the revision reaches a maximum score or (2) when the procedure performs a maximum number of steps.

Algorithm 6.8 exhibits the Hill-climbing with Stochastic Escape procedure. The first component of the main loop, which is executed when the move is stochastic, chooses a revision at random. In case its score obeys the "perturbation" constraint, it is implemented. Otherwise, the algorithm proceeds to a next iteration. The second component, the greedy move, behaves as the Stochastic Hill-climbing. Nevertheless, the algorithm has a stopping criteria differing from previous algorithms, as it stops only when a maximum score is reached or when a maximum number of iterations is executed. Such a criteria consents the procedure proceeds to a next iteration without implementing any revision, which happens when the greedy move cannot find any revision capable of improving the current score or a randomized revision degradating the score more than it is allowed.

Finally, we would like to have an algorithm which does not take greedy decisions and also accepts bad moves under a certain condition. As this idea is quite similar to what is achieved with *Simulated annealing* techniques, we implemented a version of this strategy on the base of the top-level revision algorithm.

6.4.4 Simulated Annealing Search for Revisions

Simulated annealing chooses at each iteration a candidate hypothesis at random and in case the current score is not improved, it uses a scheduling function to decide if such a hypothesis can be accepted as next hypothesis. As simulated annealing always selects a revision at random, it can be arguably faster than the previous algorithms developed in this section, since it never evaluates all the possible modifications on

Algorithm 6.8 Hill-Climbing Search for Revision with Stochastic Escape

Input: An initial theory T, A Background Knowledge FDT, a set of examples E, integer $maxSteps$, real $maxScore$
Output: A revised theory T'

1: $score \leftarrow$ compute score of T
2: $steps \leftarrow 0$
3: **repeat**
4: generate revision points
5: $possibleRevisions$ possible revisions enumerated from the revision points and respective revision operators
6: **with** probability p_{rev} **do**
7: **repeat**
8: $nextRevision \leftarrow$ a revision chosen at random from $possibleRevisions$
9: $score' \leftarrow$ compute score of $nextRevision$
10: **if** $score' + 0.5(Potentil + score')$ ¿ 0 **then**
11: $T' \leftarrow$ implements $nextRevision$
12: **else**
13: $T' \leftarrow T$
14: $possibleRevisions \leftarrow possibleRevisions - nextRevision$
15: **until** $T \neq T'$ or $possibleRevisions = \emptyset$
16: **otherwise**
17: generate all possible revisions from the revision points and respective revision operators
18: compute score $scoreNextRevision$ of each proposed revision
19: $nextRevision \leftarrow$ revision with the highest score
20: **if** $scoreNextRevision > score$ **then**
21: $T \leftarrow$ implements $nextRevision$ on T
22: $score \leftarrow scoreNextRevision$
23: $steps++$
24: **until** $score >= maxScore$ or $steps = maxSteps$

the theory. However, it can take more iterations to converge. It is proved that, if the value returned by the scheduling function is reduced slowly enough, the algorithm will find the optimal global (Russell and Norvig, 2010). Here we follow exactly the same idea and implement the simulated annealing as one of the stochastic algorithms for selecting revision to be implemented. The key components of the algorithm are as follows.

- **Defining the search space of candidate hypothesis:** Candidate hypothesis are exactly the same of the stochastic components of previous algorithms. Thus, the candidate hypothesis are all the possible revisions applied to each revision provided revision point, but without proposing them to compose the search space. Instead, they are representative tuples of each real revision, where each tuple is a revision point and a possible revision operator to be applied on it.

- **Choosing the revision to be implemented:** The revision to be implemented is chosen at random from the search space of candidate hypothesis. A proposal of the revision is generated and its score is computed. If such a score is better than the current one, the revision is implemented on the current theory. Otherwise, the revision is implemented only if a certain probability is reached. This probability is defined from the difference between the current score and the score of the revision, divided by the value computed with the scheduling function. The scheduling function, among others parameters, takes into account the number of steps performed so far. It may be the case that an iteration gets to the end without implementing any revision, precisely when a revision got at random neither can improve the current score, nor it is acceptable as a bad move.

- **Stop criteria:** The revision process stops under three conditions, namely: (1) when there are no possible revision to be implemented or (2) when the scheduling function returns zero or (3) when the theory achieves a maximum score.

Algorithm 6.9 brings Simulated strategy performed on the search for revisions. Besides the usual parameters, it requires a parameter indicating the maximum number of iterations *limit* and a reduction factor *lam*. The main loop starts by generating the revision points as usual, followed by the enumeration of tuples of possible revisions to be implemented upon such revision points. As the revisions are not really proposed before one of them has been chosen, it is necessary to check out whether it is possible to implement that revision on the theory. In possession of an implementable revision, the algorithm must decide whether it is going indeed be implemented on the current theory. To do so, first it is verified if the score can be improved after implementing the revision, which in an affirmative case, makes the revision be implemented. If the score is not improved the revision might be implemented anyway, but only when a scheduling function returns a value higher than a random probability. The scheduling function guarantees that the more iterations the algorithm performs the less is the chance of a bad move be selected. The algorithm may proceed without implementing any revision on that iteration and it finishes the revision process when the scheduling function return 0.0.

6.5 Experimental Results

In this chapter, we would like to investigate if it is possible to reduce runtime of the revision process, even when the search space is larger than usual, by the use of Stochastic Local Search techniques. Thus, the major question we would like to answer is whether the runtime of the revision process guided by SLS algorithms, can be faster than the traditional revision but without harming accuracy. A secondary question is if we can also be comparable to learning from scratch in terms of learning time, while reaching better accuracies than this approach. In this way, we have selected three ILP datasets that YAVFORTE had a hard time to revise and compared accuracy and runtime obtained from Aleph (Srinivasan, 2001b), YAVFORTE (see chapter 3) and stochastic algorithms. The datasets are as follows.

Algorithm 6.9 Simulated Annealing Search for Revisions

Input: An initial theory T, A Background Knowledge BK, a set of examples E
Output: A revised theory T'

1: $score \leftarrow$ score of T
2: $S \leftarrow 1$
3: **repeat**
4: generate revision points
5: $T' \leftarrow T$
6: $possibleRevisions$ possible revisions enumerated from the revision points and respective revision operators
7: **repeat**
8: $nextRevision \leftarrow$ a revision chosen at random from $possibleRevisions$
9: **if** it is possible implement $nextRevision$ on T **then**
10: $T' \leftarrow$ implements $nextRevision$
11: **else**
12: $possibleRevisions \leftarrow possibleRevisions - nextRevision$
13: **until** $possibleRevisions = \emptyset$ or $T' \neq T$
14: **if** $T' \neq T$ **then**
15: $scoreNextRevision \leftarrow$ compute score of T'
16: $\Delta\ E \leftarrow scoreNextRevision - score$
17: **if** $\Delta\ E > 0$ **then**
18: $T \leftarrow T'$
19: $score \leftarrow scoreNextRevision$
20: **else**
21: $S \leftarrow$ scheduling$(t, limit, lam)$
22: **if** $S \neq 0.0$ **then**
23: **with** probability $e^{\Delta\ E/S}$ **do**
24: implements $nextRevision$
25: $score \leftarrow scoreNextRevision$
26: $t++$
27: **until** $S = 0$

6.5.1 Datasets

- Pyrimidines: this is a Quantitative Structure Activity Relationships (QSAR) problem, concerning the inhibition of E. Coli Dihydrofolate Reductase by *pyrimidines*, which are antibiotics acting by inhibiting Dihydrolate Reductase, an enzyme on the pathway to forming DNA (King et al., 1992; Hirst et al., 1994a). The dataset we used in this work is composed of 2361 positive examples and 2361 negative examples.

- Proteins: This is a task of secondary structure protein prediction. The task is to learn rules to identify whether a position in a protein is in an alpha-helix (Muggleton et al., 1992). We considered a dataset with 1070 positives and 970 negatives.

- Yeast_sensitivity (Spellman et al., 1998; Kadupitige et al., 2009): This is a dataset concerning the problem of gene interaction of the *yeast Saccharomyces cerevisiae*. It is composed of 430 positive examples and 680 negative examples. It has a huge background with approximately 170,000 facts.

Experimental Methodology The datasets were splitted up into 10 disjoint folds sets to use a K-fold stratified cross validation approach. Each fold keeps the rate of original distribution of positive and negative examples (Kohavi, 1995). The significance test used was corrected paired t-test (Nadeau and Bengio, 2003), with $p < 0.05$. As stated by (Nadeau and Bengio, 2003), corrected t-test takes into account the variability due to the choice of training set and not only that due to the test examples, which could lead to gross underestimation of the variance of the cross validation estimator and to the wrong conclusion that the new algorithm is significantly better when it is not. All the experiments were run on Yap Prolog (Santos Costa, 2008).

The initial theories were obtained from Aleph system using parameters default, except for clause length, which is defined as 10 , noise, defined as 30, nodes set to 10000 and minpos, set to 2. The use of those parameters has been inspired on the work of (Muggleton et al., 2010). To generate such theories, the whole dataset was

considered but using a 10-fold cross validation procedure. Thus, a different theory was generated for each fold and each one of these theories is revised considering its respective fold (the same fold is used to generate and revise the theories). Theories returned by Aleph have about 200 clauses for each dataset, which makes them difficult for YAVFORTE to revise. Stochastic algorithms were run 5 times because of the random choices.

6.5.2 Behavior of the Stochastic Local Search Algorithms with Different Parameters

First, we would like to observe how the different stochastic strategies behave with different parameters. To do so, we used the datasets Pyrimidines and Proteins and plot curves for each individual algorithm, with different appropriate parameters.

Varying Number of Revision Points We start by comparing the revision time and accuracy results of Algorithm 6.1, which includes a stochastic component when searching for revision points, with different amounts of maximum revision points returned. Figures 6.1 and 6.2 exhibit the results for Pyrimidines and Proteins, respectively, with probability parameters fixed to 100% and 50% and number of revision points defined as 1, 5, 10 and 20. As expected, runtime increases as more revision points are returned. However, accuracies are not significantly different when 5 or more revision points are found out. Accordingly, choosing only one revision point makes the revision system extremely fast, but at the expense of worse accuracies. Accuracies are not significantly different when the probability is either 100% or 50%.

Varying Number of Iterations Algorithms Stochastic Greedy(6.6) and Hill Climbing with stochastic Escape (6.8) considers as stop criteria a maximum number of iterations. Simulated annealing also takes into account a limit to decide stopping, which is the parameter *limit* in Algorithm(6.9). In order to check the performance of these approaches when facing different maximum iterations, we fixed the probability of random walks in 100% and 50% for the two first cases and plotted the accuracy versus runtime achieved by the algorithms. The graphics are exhibited in

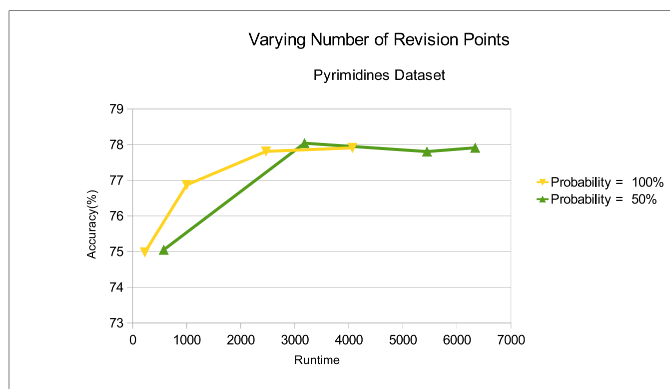

Figure 6.1: Comparing runtime and accuracy of Algorithm 6.1 in Pyrimidines Dataset, with number of revision points varying in 1, 5, 10, 20. Probabilities are fixed in 100% and 50%.

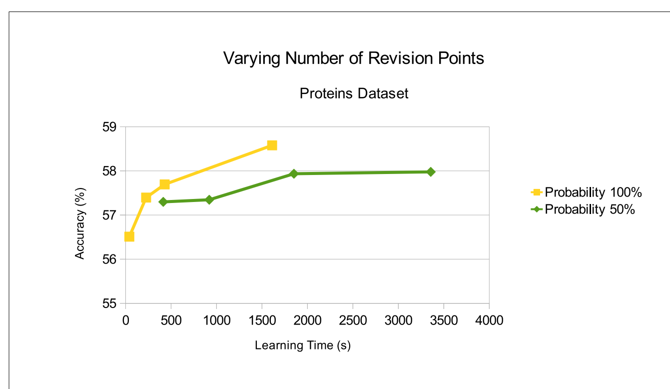

Figure 6.2: Comparing runtime and accuracy of Algorithm 6.1 in Proteins Dataset with number of revision points varying in 1, 5, 10, 20. Probabilities are fixed in 100% and 50%.

Figures 6.3 and 6.4, for Pyrimidines and Proteins datasets, respectively.

In the Pyrimidines dataset, Simulated annealing is the fastest algorithm but it only achieves accuracy equivalent to the other algorithms when the number of iterations is 40. We perform some additional tests to see if by increasing the number of iterations, simulated annealing would perform better, but we only verified higher runtime, while accuracies were stationed.

Fixing the probabilities as 100% for the two others algorithms makes them to execute very fast. However, always to perform stochastic moves does not improve

accuracies of the theories as it could. We see that by looking at the performance of these algorithms when the probability is fixed in 50%: although the revision process takes more time, since in greedy moves all revisions are evaluated so that the best one is chosen, the accuracies are also significantly higher. Hill Climbing stochastic escape algorithm with 50% of probability is the algorithm achieving best accuracies, in the same runtime as the second best algorithm, which is the Greedy Stochastic algorithm, also considering 50% of probability.

In the proteins dataset, it is interesting to see that the best accuracy was achieved by Simulated Annealing in less time than most of the cases. All the other algorithms behaves as the Pyrimidines dataset: Considering probabilities of 50% yields better accuracies, achieved in slower revision time. However, Hill climbing stochastic escape with 100% in this case is also able to achieve accuracies statistically equivalent to the 50% cases, in less time.

Those curves suggest that when increasing the number of iterations the systems are slower, as expected, but their accuracies are not significantly changed.

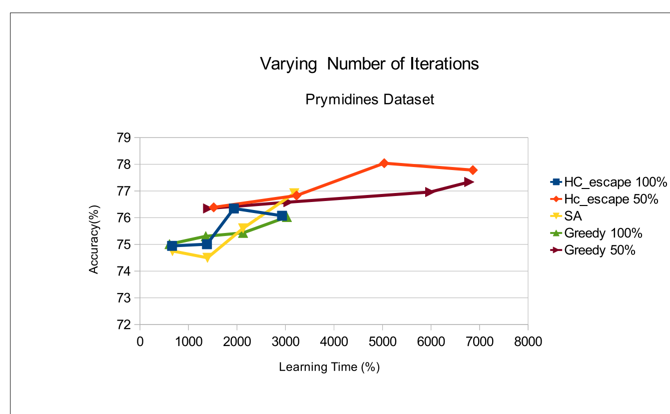

Figure 6.3: Comparing runtime and accuracy of SLS algorithms in Pyrimidines Dataset, varying maximum number of iterations, which is set to 10, 20, 30, 40. Probabilities are fixed in 100% and 50%, when it is the case.

Varying Probability Values Probability parameters are responsible for deciding the type of the move the algorithm is going to follow: either the move is greedy, and the best hypothesis found from the set of all generated hypothesis, is chosen to be

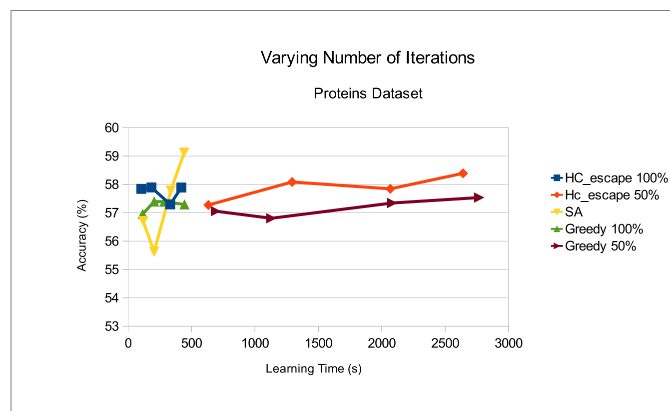

Figure 6.4: Comparing runtime and accuracy of SLS algorithms in the proteins dataset, varying maximum number of iterations, which is set to 10, 20, 30, 40. Probabilities are fixed in 100% and 50%, when it is the case.

implemented, or the move is stochastic and possible hypothesis are randomized. All algorithms but Simulated Annealing follow this strategy. To see the performance of the algorithms when facing different probability parameters, we plot accuracy versus runtime curves for each algorithms, considering probabilities as 100%, 80%, 60% and 40%. Maximum number of iterations for Hill climbing with stochastic escape and Stochastic Greedy algorithms for choosing revisions are set to 20, since in previous section we see this value has the best balance between accuracy and runtime. Maximum number of revision points is set to 10.

In Pyrimidines dataset, the performance of Greedy and Stochastic Escape algorithms follow the same pattern: the smaller is the probability, higher accuracies are achieved at the expense of higher runtime. However, while Stochastic Escape can reach accuracy very close to the other algorithms, in the best case, accuracies of Greedy algorithm are always significantly slower than the other algorithms. The accuracies achieved by the rest of the algorithms slightly changes, but they are not significantly different, no matter the probability value is. On the other hand, the smaller is the chance of choosing a stochastic move, the slower is the revision process. It is interesting that in these cases, even always following a stochastic move (probability set to 100%), accuracies are not significantly affected. Thus, we can conclude from this dataset that, if the stochastic move does not demands

an improvement on the score, which is the case of Greedy and Stochastic Escape, probabilities play a fundamental role. On the other hand, if both moves require improvement on the score (stochastic Hill climbing), high probabilities can yield the same results as low probabilities, but in less time.

Proteins dataset has similar results. In this case, the stochastic hill climbing search for revisions and stochastic search for literals achieve the best accuracy results, but at the expense of higher runtime.

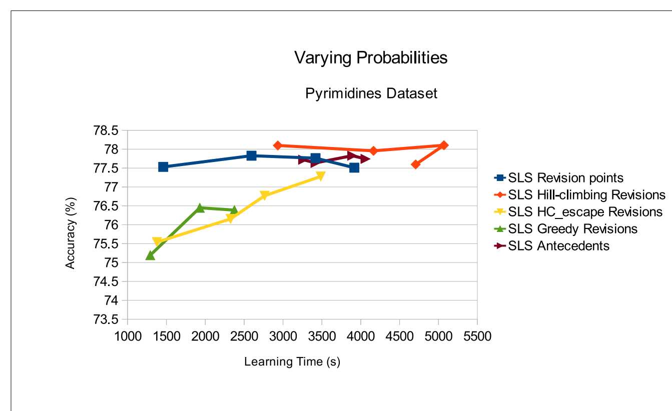

Figure 6.5: Comparing runtime and accuracy of SLS algorithms in Pyrimidines Dataset, with probabilities varying in 100, 80, 60, 40. Maximum number of iterations is fixed in 20, when it is the case.

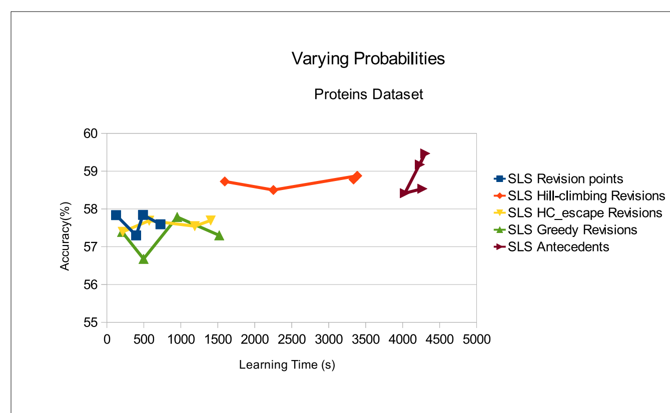

Figure 6.6: Comparing runtime and accuracy of SLS algorithms in Proteins Dataset, with probabilities varying in 100, 80, 60, 40. Maximum number of iterations is fixed in 20, when it is the case.

6.5.3 Comparing Runtime and Accuracy of SLS Algorithms to Aleph and YAVFORTE

In this section we would like to compare the performance of Stochastic Local Search Algorithms to state-of-the-art learning from scratch and revision systems. We compare the accuracy and runtime of Aleph and YAVFORTE (see chapter 3) to the SLS algorithms presenting a better trade-off between accuracy and runtime. In addition to Pyrimidines and Proteins, we also consider here the Yeast sensitivity dataset, but in this case, concerning the search for revisions, we show the results only for stochastic Hill climbing, since this has the best accuracy results for the other two datasets. Besides running individually the stochastic algorithms for each key search, we also combine stochastic algorithms as follows.

1. Stochastic search for revision points with stochastic search for literals.
2. Stochastic search for revision points with stochastic Hill climbing search for revisions.
3. Stochastic Hill climbing search for revisions with stochastic search for literals.
4. Stochastic search for revision points with stochastic search for literals and stochastic Hill climbing search for revisions.

Figures 6.7, 6.8 and 6.8 presents the accuracies returned by the systems for Pyrimidines, Proteins and Yeast sensitivity datasets, respectively. The parameters used to each system are presented in parenthesis, where the first value is a probability parameter and the second value (when it is the case) it is the number of iterations/revision points. Next, we discuss the most relevant cases observed in the graphic.

- All revision algorithms in the three datasets achieve accuracies significantly better than Aleph system.
- Except for Greedy Search for revisions, in Pyrimidines dataset there is no statistical significant difference between the accuracy returned by YAVFORTE and the stochastic algorithms.

- Best accuracy result for Pyrimidines is achieved by stochastic search for literals combined with Hill climbing stochastic search for revisions, although it is not significantly better than the others.

- In Proteins dataset, best results are achieved by stochastic search for literals and stochastic search for literals combined with Hill climbing stochastic search for revisions.

- In Proteins dataset, stochastic search for revision points and stochastic greedy search for revisions achieves significantly worse accuracies, compared to YAVFORTE and best SLS results.

- There is no significant difference between YAVFORTE and the rest of the SLS algorithms.

- In Yeast Sensitivity dataset, stochastic search for literals and the combination of the three stochastic components achieve significantly better accuracies than YAVFORTE.

- The other SLS algorithms do no present significant difference compared to YAVFORTE

From these results, we can see that SLS algorithms either provide better or equivalent accuracies compared to the baseline revision system YAVFORTE. In addition, they are always significantly better than Aleph system.

Figures 6.10, 6.11 and 6.12 exhibit runtime of Aleph, YAVFORTE and SLS algorithms for Pyrimidines, Proteins and Yeast sensitivity, respectively. The following issues are observed from the results graphically represented in the figures.

- Pyrimidines Dataset
  - Revision time of YAVFORTE is significantly slower than all other algorithms in Pyrimidines dataset.
  - In Pyrimidines dataset, the only algorithm significantly slower than Aleph is SLS with revisions escape. This is likely due to the high number of iterations.

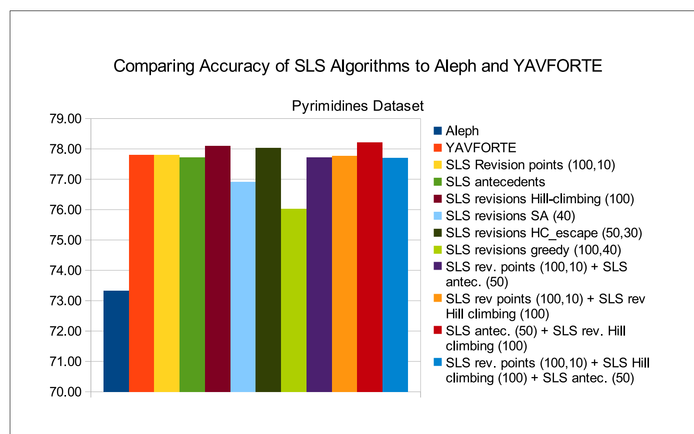

Figure 6.7: Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Pyrimidines Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

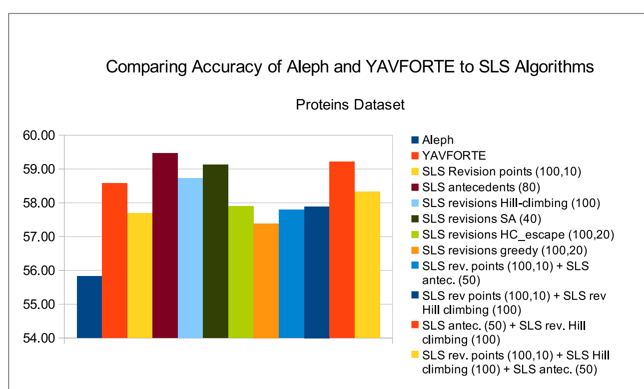

Figure 6.8: Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Proteins Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

– Stochastic Hill climbing search for revisions combined with stochastic search for literals does not present difference compared to Aleph. In this case, the combination of these SLS algorithms takes more time to converge.

– All the other SLS cases are significantly faster than Aleph.

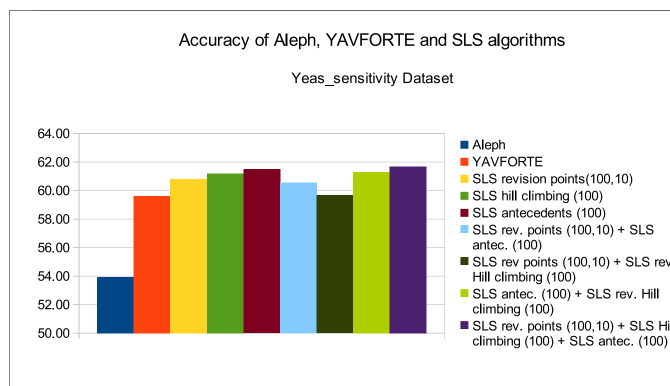

Figure 6.9: Comparing accuracy of SLS algorithms to Aleph and YAVFORTE in Yeast Sensitivity Dataset. Results of SLS algorithms are the ones with a best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

  – Higher speed of a SLS algorithm, compared to YAVFORTE has a factor of 17, while smallest speed up is 4.

- In Proteins dataset all SLS algorithms are significantly faster than Aleph and YAVFORTE, except for stochastic search for literals, which is faster than YAVFORTE but it is not significantly different than Aleph. Higher speed of a SLS algorithm, compared to YAVFORTE has a factor of 16, while smallest speed up is 2.

- In Yeast sensitivity, Aleph is significantly faster than all revision algorithms. We believe this is due to the huge background knowledge this dataset has, that makes coverage tests slower and revision must perform much more such kind of tests than learning from scratch approaches. On the other hand, all SLS algorithms are significantly faster than YAVFORTE. Higher speed of a SLS algorithm, compared to YAVFORTE has a factor of 26, while smallest speed up is 2.

From the results, we can conclude that using stochastic local search algorithms individually and specially combining them, it is possible to greatly reduce the revision time and also to be faster or competitive with learning from scratch. Moreover, accuracies achieved by SLS strategies are always better than learning from scratch

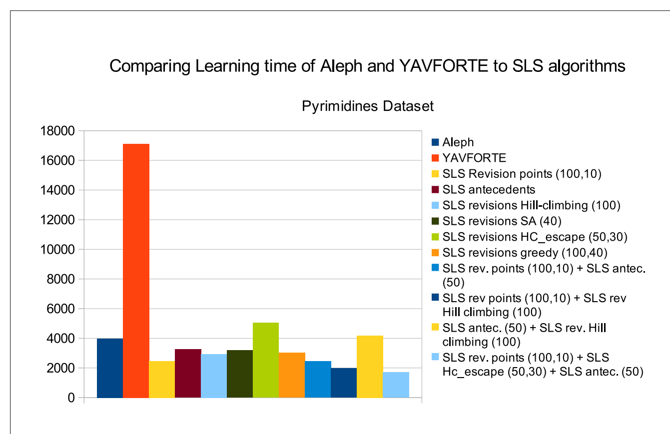

Figure 6.10: Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Pyrimidines Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

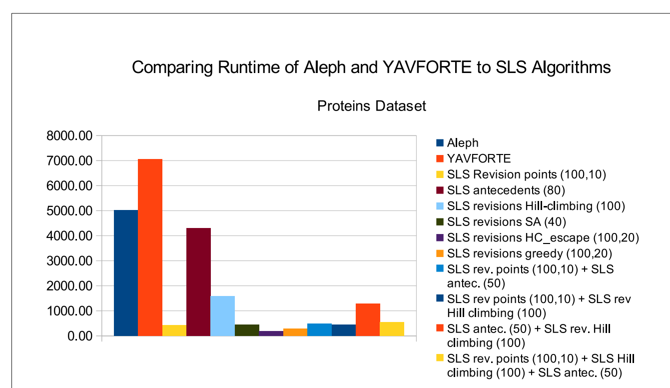

Figure 6.11: Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Proteins Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

approach and competitive with traditional revision approach. Thus, we positively answer both questions posed in the beginning of this section.

6.6 Conclusions

In this chapter we designed a set of stochastic local search algorithms for exploring the key search spaces of the revision process more efficiently. The algorithms aban-

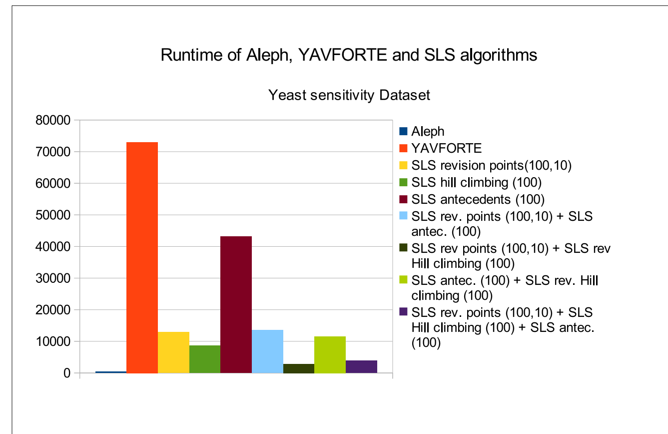

Figure 6.12: Comparing runtime of SLS algorithms to Aleph and YAVFORTE in Yeast Sensitivity Dataset. Results of SLS algorithms are the ones with the best trade-off between accuracy and runtime. Parameters of probability, number of iterations and number of revision points are in parenthesis.

don completeness in favor of finding good solutions in a reasonable time. Most of all are based on random walks, so that the choice of pursuing a greedy or a stochastic move is made according to a probability parameter. Stochastic algorithms were implemented in YAVFORTE system (see chapter 3 and (Duboc et al., 2009)) in every key search of the revision process.

First, a SLS algorithm was built to avoid collecting all revision points from all misclassified examples. With a probability p, misclassified examples are randomized and a pre-defined number of revision points is generated. The search is alternated with greedy moves, since without the probability p all revision points found in the theory from each misclassified instance are collected. However, through experimental results we found out that in several cases there is no need of employing greedy moves, since defining the probability parameter as 100% already achieves good results: the revision time is greatly reduced and accuracies are not statistically significant different compared to the baseline system. The performance of stochastic component in the search for revision points is more influenced by the parameter defining the number of revision points that should be returned.

Second, SLS components were included in the search for literals to be added to or removed from a clause. Stochastic search was included in both hill climbing

and relational pathfinding algorithms for specializing clauses. In the first case, when the move is stochastic, literals from the Bottom Clause are randomized and the first literal found that improve the score is added to the clause. In relational pathfinding, paths created from the Bottom Clause and a positive instance are randomized. In this work, we used both algorithms cooperating to each other, when relational pathfinding was applicable. We noticed from the empirical evaluation that, although the stochastic search for literals is able to reduce runtime compared to hill climbing greedy approach of YAVFORTE, if it is executed without the other stochastic components, it is not much effective as the other SLS algorithms. The reasons for that are mainly due to the Bottom clause: either it is small and the stochastic component does not make a huge difference, or it is large, but in this case the use of stochastic move takes more iterations to converge than the original approach. Novel approaches can be investigated to further improve this stochastic component, such as pre-process literals using Bayesian networks or genetic algorithms (Oliphant and Shavlik, 2008; Muggleton and Tamaddoni-Nezhad, 2008; Pitangui and Zaverucha, 2011).

Third, four different stochastic components were included to decide which revision is going to be implemented. Three of them are based on random walks and one of them performs a simulated annealing algorithm. In the results we could see that the stochastic greedy algorithm for searching revisions, which in both stochastic and greedy moves allows a revision with score worse than the current one be implemented, does not performs well. As an improvement in score is always not required, the accuracy deteriorates along the iterations. The stochastic escape approach, that accepts bad moves but only if it does not degrade so much the score, performs better than greedy, with good accuracy and runtime results in several cases. However, the simplest strategy, that with a certain probability randomizes revisions and implements the first one improving the score, achieved the best overall results.

The three strategies above were compared both individually and combining the best strategies. The vast majority of the cases showed that stochastic approaches achieve better accuracies than the learning from scratch Aleph system, with faster or competitive runtime. Moreover, accuracies in most of the cases are significantly

equivalent to YAVFORTE but the runtime is always much faster. In the best case, an SLS algorithm is $25X$ faster than YAVFORTE, with equivalent accuracy.

We see at least two different topics for further investigation concerning SLS in first-order logic revision: the use of rapid randomized restart strategies, already employed in ILP in (Železný et al., 2004; Železný et al., 2006), that would avoid being stuck in unproductive refinements by restarting after a certain criteria from another random point; and the use of stochastic component in coverage tests, as it is done in (Sebag and Rouveirol, 1997; Kuzelka and Zelezný, 2008a).

Chapter 7

Probabilistic Logic Learning

Traditional statistical machine learning algorithms deal with uncertainty by assuming the data are independent and identically distributed (i.i.d). On the other hand, relational learning algorithms have the capability to represent multi-typed related objects, but they impose severe limitations for representing and reasoning in the presence of uncertainty. However, in most real-world applications, data is multi-relational, heterogeneous, uncertain and noisy. Examples include data from web, bibliographic datasets, social network analysis, chemical and biological data, robot mapping, natural language, among others (Lachiche and Flach, 2002; Battle et al., 2004; Getoor et al., 2004; Jaimovich et al., 2005; Davis et al., 2005a; Neville et al., 2005; Wang and Domingos, 2008; Raghavan et al., 2010). Therefore, in order to extract all useful information from those datasets it is necessary to use techniques dealing with multi-relational representations and probabilistic reasoning. *Probabilistic Logic Learning (PLL)*, also called *Probabilistic Inductive Logic Programming (PILP)* (De Raedt et al., 2008a) and *Statistical Relational Learning (SRL)* (Getoor and Taskar, 2007) is an emerging area of artificial intelligence, lying at the intersection of reasoning about uncertainty, machine learning and logical knowledge representation (De Raedt, 2008). As such, PLL is able to deal with machine learning and data mining in complex relational domains where information may be missed, partially observed and/or noisy.

A large number of PLL systems have been proposed in the last years, giving rise to several formalisms for representing logical and probabilistic knowledge. The formalisms can be divided into several general classes (Getoor, 2007), yet here we

choose to put them in the two more relevant axes (De Raedt et al., 2008a): *Logical Probabilistic Models (LPM)* are extensions of probabilistic models that are able to deal with objects and relations by including logical or relational elements. Typically, they build a directed or undirected probabilistic graphical model to reasoning about uncertainty using logic as a template. Inside this category there is a group of formalisms based upon directed probabilistic graphical models, composed of Relational Bayesian Networks (Jaeger, 1997), Probabilistic Logic Programs (Ngo and Haddawy, 1997; Haddaway, 1999), Probabilistic Relational Models (PRM) (Koller and Pfeffer, 1998; Koller, 1999; Friedman et al., 1999; Getoor et al., 2001), *Bayesian Logic Program (BLP)* (Kersting and De Raedt, 2001d; Kersting and De Raedt, 2001b; Kersting and De Raedt, 2001a; Kersting and De Raedt, 2007), Constraint Logic Programming with Bayes Nets (CLP(BN)) (Santos Costa et al., 2003a), Hierarchical Bayesian Networks (Gyftodimos and Flach, 2004), Logical Bayesian Networks (Fierens et al., 2005), Probabilistic Relational Language (Getoor and Grant, 2006), etc, and a group composed of systems built upon undirected probabilistic graphical models, including Relational Markov Networks (Taskar et al., 2002), Relational Dependency Networks (Neville and Jensen, 2004)and Markov Logic Networks (MLN) (Singla and Domingos, 2005; Kok and Domingos, 2005; Richardson and Domingos, 2006; Domingos and Lowd, 2009).

In the other axis are the *Probabilistic Logical Models (PLM)*, which are formalisms extending logic programs with probabilities, staying as close as possible to logic programming by annotating clauses with probabilities. In this class, the logical inference is modified to deal with the parameters of probabilities. Formalisms following this approach include Probabilistic Horn Abduction (Poole, 1993) and its extension Independent Choice Logic (ICL) (Poole, 1997), *Stochastic Logic Program (SLP)* (Muggleton, 1996; Muggleton, 2000; Muggleton, 2002), PRISM (Sato and Kameya, 1997; Sato and Kameya, 2001), Logic Programs with Annotated Disjunctions (Vennekens et al., 2004), SAYU (Davis et al., 2005a; Davis et al., 2005b; Davis et al., 2007), nFoil (Landwehr et al., 2007), kFoil (Landwehr et al., 2006), *ProbLog* (De Raedt et al., 2007; Kimmig et al., 2008; Kimmig, 2010), among others.

The present work aims to contribute to the rule-based formalism using the

directed probabilistic graphical model of Bayesian networks. BLP was chosen to conduct the research of theory revision in PLL since it elegantly instantiates both logic programs and Bayesian networks. Next we bring some details on the fundamentals of BLPs, starting from Bayesian Networks in section 7.1, immediately followed by the ideas behind BLPs in section 7.2. Finally, we review our revision system PFORTE in section 7.3.

For some basic concepts from probability theory we refer to (Feller, 1970; Ross, 1988; Pearl, 1988; DeGroot, 1989).

7.1 Bayesian Networks: Key Concepts

Complex systems involving some sort of uncertainty may be characterized through multiple interrelated random variables, where the value of each variable defines an important property of the domain. In order to probabilistically reason about the values of one or more variables, possibly given evidence about others, it is necessary to construct a joint distribution over the space of possible assignments to a set of random variables. Unfortunately, even in the simplest case of binary-valued variables, the representation of a joint distribution over a set of not assumed independent random variables $\chi = X_1, ..., X_n$ requires the specification of the probabilities of at least 2^n different assignment of values $x_1, ..., x_n$. It is obviously unmanageable to explicitly represent such a joint distribution.

Probabilistic Graphical Models (PGMs) provide mechanisms for encoding such high-dimensional distributions over a set of random variables, structuring them compactly so that the joint distribution can be utilized effectively (Koller and Friedman, 2009). They use a graph-based representation, where the nodes correspond to the random variables in the domain and the edges correspond to direct probabilistic interactions between the variables.

There are two most common used families of PGM, one representing the domain through undirected graphs and the other through directed graphs. In the first case lies the *Markov networks* (aka *Markov random field*), consisting of an undirected graph G and a set of potential functions ϕ_k (Pearl, 1988). The graph has a node for each random variable and the model has a non-negative real-valued

potential function for each clique in the graph. Markov networks are useful in modeling problems where one cannot naturally define a directionality to the interactions between the variables. The representative of the second case are the Bayesian networks (Pearl, 1991) whose edges of the graph have a source and a target. Next we describe Bayesian networks thoroughly, since it is the underlying probabilistic graphical models of BLPs.

In order to review the main concepts of Bayesian networks we use the following convention: X denotes a random variable, x a state, $\mathbf{X}$ a set of random variables and $\mathbf{x}$ a set of states.

Bayesian networks represent the joint probability distribution $P(X_i, ..., X_n)$ over a fixed and finite set of random variables $\{X_i, ..., X_n\}$. Each random variable X_i has a $domain(X_i)$ of mutually exclusive states. They allow a compact and natural representation of the set of random variables by exploiting conditional independence properties of the distribution.

We say a variable X is conditionally independent of a variable Y, given a variable E, in a distribution P if $P(X|Y, E) = P(X|E)$. Conditional independence is denoted by $(X \perp Y|Z)$.

To represent the connections between random variables, as well as their probability distributions, a Bayesian network is composed of two components, as follows. *Qualitative or logic component of a Bayesian network*: it is an augmented Directed Acyclic Graph (DAG) G whose nodes are the random variables in the domain and whose edges correspond to direct influence among the random variables. The *parents* of a variable X_i are the variables represented by the nodes whose edges arrive in X_i. Similarly, the *children* of a variable X_j are variables represented by the nodes whose edges come from X_j. Henceforward, we use the terms “variables ” and “nodes ” interchangeably. The *local independence assumption* in a Bayesian network states that a variable X_i is conditionally independent of its non-descendants in the network, given a joint state of its parents, i.e.:

$$(X_i \perp Non - Descendants_{X_i}|Parents(X_i)) \tag{7.1}$$

where $Parents(X_i)$ denotes the states of the parents of node X_i, and if the

node has no parents, then $P(X_i|Parents(X_i) = P(X_i)$

The independence assumption allows it to write down the joint probability distribution as

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|\ Parents(X_i)) \tag{7.2}$$

by applying the independence assumption to the *chain rule* expression of the joint probability distribution.

Quantitative component of a Bayesian network. Because of the independence assumption, each node is associated to a *local probability model* that represents the nature of the dependence of each variable on its parents. Thus, each node has a Conditional Probability Distributions (CPDs), $cpdX_i$, specifying a distribution over the possible values of the variable given each possible joint assignment of values to its parents, i.e, $P(X_i|parents(X_i)$. If a node has no parents, then the CPD turns into a marginal or prior distribution, since is conditioned on an empty set of variables.

Consider, for instance the following problem from genetics (Friedman et al., 1999; Kersting et al., 2006): *It is a genetic model of the inheritance of a single gene that determines the blood type of a person. Each person has two copies of the chromosome containing this gene, one inherited from her mother and another inherited from her father. Occasionally, a person is not available for testing, and yet because of the clarification of crime, test of paternity, allocation of (frozen) semen etc. it is often necessary to estimate the blood type of the person. A blood type can still be derived for that person through an examination and analysis of the types of family members.*

To represent this domain we would have for each person three random variables: one representing her blood type (bt_{person}), another one representing the gene inherited from her father (pc_{person}) and the last one representing the gene inherited from her mother (mc_{person}). The possible values for bt_{person} are in the set $domain(bt_{person}) = \{a, b, o, ab\}$ and the domain for mc and pc are the same and composed of $\{a, b, o\}$. In this example the independence assumptions are clear due to the biological rules: once we know the blood type of a person, additional evidence about others members of the family will not provide new information about the blood type. In the same way, once we know the blood type of both parents,

we know what each of them can pass on to their descendants. Thus, finding new information about a person's non-descendants does not provide new information about blood type of the person's children. Those local dependencies are better visualized through the Bayesian network of the Figure 7.1. For example, regarding the biological point of view, the blood type of *Susan* is correlated with the blood type of her aunt *Lily*, but once we know the blood type of *Allen* and *Brian*, the blood type of *Lily* is not going to be relevant to the blood type of *Susan*. Each variable has an associated CPD, coding the probability of a person has one of the values of the domain as her blood type, given each possible assignment of the genes inherited from her parents.

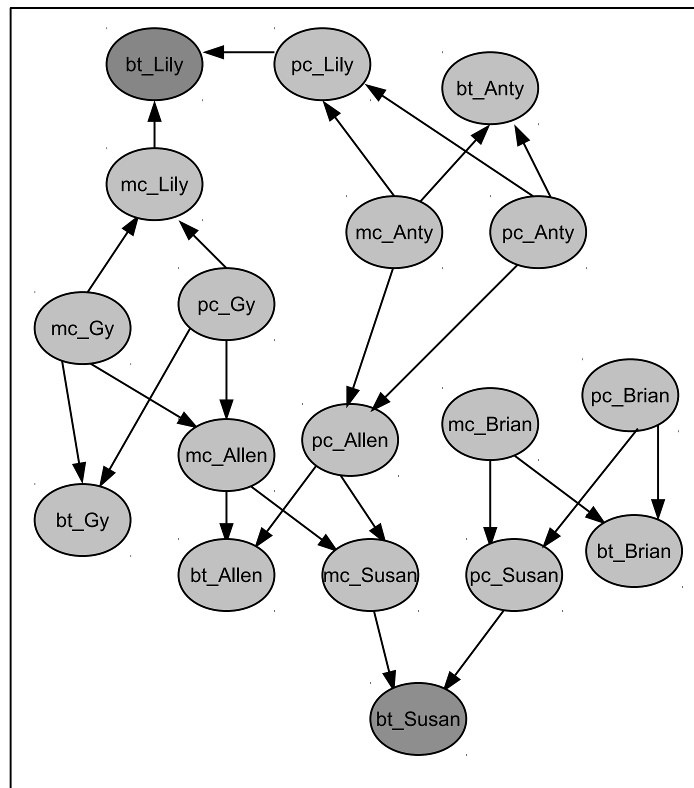

Figure 7.1: Bayesian network representing the blood type domain within a particular family

7.1.1 D-separation

Independence properties in probabilistic graphical modes can be exploited in order to reduce the computation cost of answering queries process. If one guarantees that

a set of nodes $\mathbf{X}$ is independent from another set of nodes $\mathbf{Y}$ given $\mathbf{E}$, then in the presence of evidence about the nodes in $\mathbf{E}$, an observation regarding variables in $\mathbf{X}$ cannot *influence* the beliefs about $\mathbf{Y}$. The independence assumptions satisfied by a Bayesian network can be efficiently identified using a graphical test called *d-separation* (Geiger et al., 1989). The intuition above this test encompasses four general cases, illustrated in Figure 7.2, from where we would like to analyze whether knowing an evidence about a variable X can change the beliefs about a variable Y, in the presence of evidence about variables $\mathbf{E}$. Naturally, when variables X and Y are directly connected, they are able to influence each other. Now, consider the four cases where variables X and Y are connected through E.

- The first case represents an indirect causal effect, where an ancestor X of Y could pass influence to it via $\mathbf{E}$. Consider, for example, the Bayesian network of Figure 7.1. If one wants to know the blood type of *Susan* and do not know the gene she receives from her mother (*mc_Susan* is unknown), the gene her mother received from Susan' grandmother (represented by the known variable *mc_Allen*) is able to influence the beliefs on which blood type Susan is. On the other hand, if one already knows the gene Susan received from her mother, the gene passed from Susan'grandmother to Susan' mother no longer influences her blood type. Referring to the general case, X can only influence Y in the presence of E if E is not observed. In this case, we say the evidence E *blocks* influence of X over Y, as stated in Definition 7.1.

- The next case is the symmetrical case of the last one: we want to know whether evidence above a descendant may affect an indirect ancestor. In our running example, this is the case of trying to know whether the gene *Allen* received from her mother (random variable *mc_Allen*) is affected by the knowledge of the blood type of Susan (*bt_Susan*). As before, once the gene Susan received from Allen is known (random variable *mc_Susan* is observed), *bt_Susan* is not able to affect the beliefs on *mc_Allen*. However, in case *mc_Susan* is missing, *bt_Susan* has a free path to reach *mc_Allen* and therefore to affect its value.

- A common cause case is represented in Figure 7.2(c), where the node E is a

common cause to nodes X and Y. In the Bayesian network of the example, this is the case of variables *mc_Allen*, *mc_Susan* and *bt_Allen*, where *mc_Allen* is a common cause for the others variables. The bloodtype of *Allen* (*bt_Allens*) and the gene *Susan* received from her mother *Allen* (*mc_Susan*) are correlated, since knowing the bloodtype of *Allen* helps us to predict the gene she received from her mother (*mc_Allen*) and consequently, the gene *Allen* has transmitted to her daughter *Susan*. However, in case *mc_Allen* is observed, knowing the bloodtype of *Allen* provides no additional information to predict the gene she transmitted for her daughter, as the information of which gene she has is stronger than her bloodtype. Thus, in the general case, variables sharing a common cause are able to influence each other if the path of common causes between them if free of evidences, i.e, the path is unblocked.

- Figure 7.2(d) represents a common effect trail. Structures in the form $X \rightarrow E \leftarrow Y$ are called *v-structures*. The three cases previously discussed share a pattern: X can influence Y via E is and only if E is not observed. In constrast, a common effect trail is different as X can influence Y via E if and only if E *is* observed. This is easier to see through an example: consider the random variables *mc_Susan*, *pc_Susan* and *bt_Susan* in Figure 7.1 and suppose we would want to know the gene *Susan* received from her father (random variable *pc_Susan*). If the blood type of *Susan* is known, the evidence on *mc_Susan* is able to influence the beliefs on *pc_Susan*. Knowing the gene she received from her mother *and* her blood type affects the beliefs about the gene she received from her father. For example, knowing her blood type is A and the gene her mother transmitted to her is o, the only possible gene she received from his father is A. However, if we do not know the blood type of *Susan*, it is not possible that the gene she received from her mother affects our beliefs on the gene she received from her father. Thus, if the common effect variable is not observed, knowing about a parent variable cannot affect our expectation about the others parents.

Definition 7.1 *Block: Let X and Y be random variables in the graph of a Bayesian*

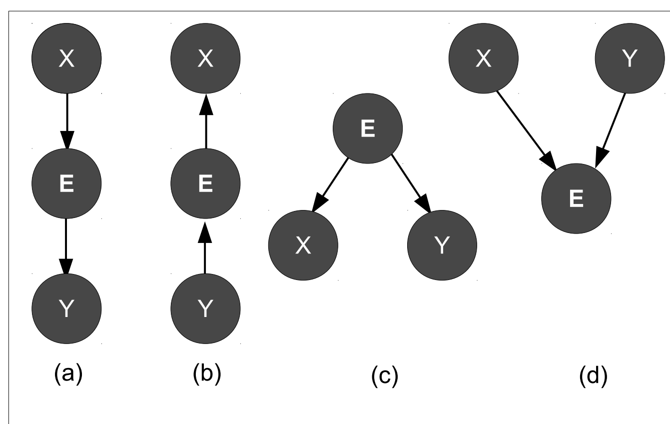

Figure 7.2: The four possible edge trails from X to Y, via E. Figure (a) is an indirect causal effect; Figure (b) is an indirect evidential effect; Figure (c) represents a common cause between two nodes and Figure (d) shows a common effect trail.

network. We say an undirected path between X and Y is blocked by a (set of) variable E if E is in such a path and influence of X cannot reach Y and change the beliefs about it because of evidence (or lack of it) of E.

The d-separation test guarantees that **X** and **Y** are independent, given **E** if every path between **X** and **Y** is blocked by **E** and therefore influence cannot flow from **X** through **E** to affect the beliefs about **Y** (Koller and Friedman, 2009; Darwiche, 2010). If the path is not blocked, we say there is an *active trail* between the two sets of nodes.

Definition 7.2 *Active trail: Given an undirected path $X_1 \rightleftharpoons \ldots \leftrightharpoons X_n$ in the graph component of a Bayesian network, there is an* active trail *from X_1 to X_n given a subset of the observed variables* **E**, *if*

- *whenever we have a v-structure $X_{i-1} \rightarrow X_i \leftarrow X_{i+1}$, then X_i or one of its descendants are in* **E***;*
- *in all the others cases no other node along the trail is in* **E***.*

An active trail indicates precisely a path in the graph where influence can flow from one node to another one. Thus, we say that one node can influence another if there is any active trail between them. The definition below provides the notion of separation between nodes in a directed graph.

Definition 7.3 *d-separation:* $\mathbf{X}$ *and* $\mathbf{Y}$ *are* d-separated *by a set of evidence variables* $\mathbf{E}$ *if there is no active trail between any node* $X \in \mathbf{X}$ *and* $Y \in \mathbf{Y}$ *given* $\mathbf{E}$*, i.e, every undirected path from* $\mathbf{X}$ *to* $\mathbf{Y}$ *is blocked.*

Consider, for example the network in Figure 7.1, and the random variables $bt(Lily)$ and $bt(Susan)$. There is a path between them composed of the nodes

$$bt(Lily) \leftarrow mc(Lily) \leftarrow mc(Gy) \rightarrow mc(Allen) \rightarrow mc(Susan) \rightarrow bt(Susan)$$

that is an active trail: the path is a common cause trail, and there is no observed node between them blocking the influence of $bt(Lily)$ over $bt(Susan)$. Note, however, in case there is an observed node in the path between them, say, $mc(Susan)$, the path would be no longer an active trail, since this evidence would block the influence of $bt(Lily)$.

It has been proved that $\mathbf{X} \perp \mathbf{Y}|\mathbf{E}$ if and only if $\mathbf{E}$ d-separates $\mathbf{X}$ from $\mathbf{Y}$ in the graph G (Geiger et al., 1989; Geiger et al., 1990).

Bayes Ball

In order to answer a probabilistic query more efficiently, it is useful to identify the minimal set of relevant random variables, which are the ones influencing the computations. (Shachter, 1998) developed a linear time algorithm to identify conditional independence and requisite information in a Bayesian network, named the *Bayes Ball* algorithm, based on the concept of D-separation. Requisite information is composed of the nodes for which conditional probability distributions or observations might be needed to compute the probability of query nodes, given evidence. Bayes Ball works by using an analogy of bouncing a ball to visit the relevant nodes in the net, starting from the query nodes. From there, the ball may pass through the node from one of its parents to its children and vice-verse, may bounce back from any parent to all the others parents of the node or from any child to all children of the node or may be blocked. The move the ball takes also depends whether the node is observed or not and/or whether the node is deterministic or not. Precisely,

- An observed node, deterministic or not, always bounces balls back from one parent to all its others parents, in order to identify common effect trails. How-

ever, such a node blocks balls from children, i.e., if a ball comes from one of its children it is not passed anymore from this node.

- An unobserved non deterministic node passes balls from parents to its children and from a child to both its parents and to others children.
- An unobserved deterministic node always passes the ball coming from its child to its parents and from a parent to its children.

Algorithm 7.1 formalizes what we have just said, with some important additions to guarantee that the same action is not repeated and that the algorithm finishes:

- It marks visited nodes on the top(bottom) when the ball is passed from a node to its parents(children). When a node is marked in the top (bottom) the algorithm has no need to visit the node's parent (children) anymore.
- It maintains a schedule of nodes to be visited from parents and from children, so that the move of the ball can be determined;
- It makes sure that the ball visits the same arc in the same direction only once.

After visiting all scheduled nodes, the algorithm returns as the minimum set of requisite information to answer a query $P(\mathbf{X}|\mathbf{E})$ the observed nodes which were visited and all residual nodes marked on the top, with the visit starting from the nodes in $\mathbf{X}$. It also identifies the nodes not marked on the bottom as irrelevant to estimate X, that is those nodes conditionally independent on $\mathbf{X}$, given $\mathbf{E}$.

It is proved that $\mathbf{X} \perp \mathbf{Y}|\mathbf{E}$ if and only if $\mathbf{Y} \subseteq \mathbf{I}$, where I is the set of conditionally independent nodes from $\mathbf{X}$ given $\mathbf{E}$, as determined by Algorithm 7.1. As any edge is traversed at most once in each direction, the complexity of the algorithm is $O(n+m)$, where n is the number of nodes and m is the number of edges in the graph.

Figure 7.3 shows an example of the execution of Bayes Ball algorithm in a Bayesian network extracted from Figure 7.1. The visit starts from the random variable bt_Susan, as it had been visited before from a child. Next,

Algorithm 7.1 The Bayes Ball Algorithm to Collect Requisite Nodes in a Network (Shachter, 1998)

Input: a Bayesian network B, a set $\mathbf{F}$ of deterministic nodes; A set $\mathbf{X}$ of query nodes; A set $\mathbf{E}$ of observed nodes

Output: A minimum set $\mathbf{R}$ of requisite nodes, which might be needed to compute $P(\mathbf{X}|\mathbf{E})$ and the set $\mathbf{I}$ of irrelevant nodes, where $\mathbf{X} \perp \mathbf{I}|\mathbf{E}$

1: Initialize all nodes in B as neither visited, nor marked on the top, nor marked on the bottom.
2: Create a schedule of nodes to be visited, initialized with each node in $\mathbf{X}$ to be visited as if from one of its children.
3: **while** there are still nodes scheduled to be visited **do**
4: Pick any node K scheduled to be visited and remove it from the schedule. Either K was scheduled for a visit from a parent, a visit from a child, or both.
5: Mark K as visited.
6: **if** $K \notin \mathbf{E}$ and the visit is from a child **then**
7: **if** the top of K is not marked **then**
8: mark its top and schedule each of its parents to be visited;
9: **if** $K \notin \mathbf{F}$ and the bottom of K is not marked **then**
10: then mark its bottom and schedule each of its children to be visited.
11: **if** the visit to K is from a parent **then**
12: **if** $K \in \mathbf{E}$ and the top of K is not marked **then**
13: mark its top and schedule each of its parents to be visited;
14: **if** $K \notin \mathbf{E}$ and the bottom of K is not marked **then**
15: mark its bottom and schedule each of its children to be visited.
16: $R \leftarrow$ nodes in $\mathbf{E}$ marked as visited $\cup$ nodes marked on top
17: $I \leftarrow$ nodes not marked on the bottom
18: **return** R and I

- As *bt_Susan* is not observed, it is marked in the bottom, in the top and passes the ball to its parents. Then, nodes *mc_Susan* and *pc_Susan* are scheduled to be visited. This corresponds to lines 6 – 10 of Algorithm 7.1.
- Node *pc_Susan* is picked in the schedule. As it is an observed node, visited from a child, the ball is not passed anymore from it. In another words, its evidence blocks influence to its child that could pass through it.
- Node *mc_Susan* is marked in the bottom and in the top and passes the ball from its child to both its parents (*mc_Allen* and *pc_Allen*) and its child (*bt_Susan*), since it is not observed.

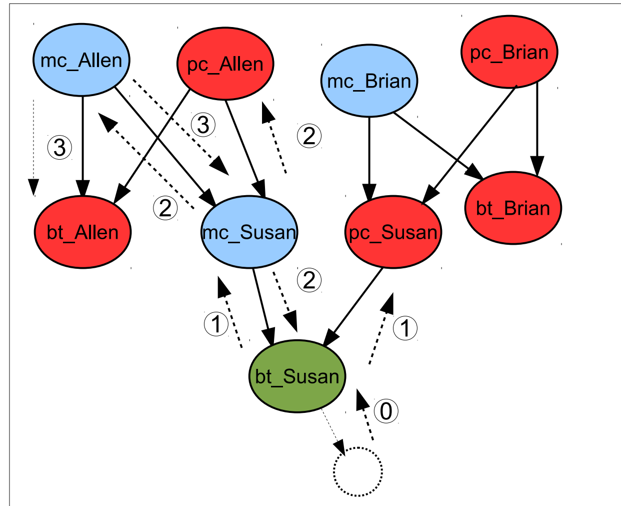

Figure 7.3: The ball visiting nodes with Bayes Ball algorithm. Red nodes are observed and green node is the query node.

- Node *bt_Susan* is visited again, this time from its parent, but there is nothing further to do, since it has already been marked in both directions.
- Node *pc_Allen* receives the ball, but it does not pass the ball anymore, since it is an observed node visited from a child.
- Node *mc_Allen* is marked in the top and in the bottom and bounces the ball to its children, *mc_Susan* and *bt_Allen*.
- Node *mc_Susan* is visited again, this time from its parent, but there is nothing further to do, since it has already been marked in both directions (bottom and top).
- *bt_Allen* in an observed node, visited from a parent, and therefore is marked in the top (line 13 of the algorithm). Then, it bounces the ball back to all its parents, in an attempt to find out a common effect trail between its parents. *mc_Allen* is marked in both directions: there is nothing more to do with it. *pc_Allen* is again visited from a child. As there is no nodes scheduled to be visited anymore, the algorithm finishes.

Requisite nodes to compute the probability of *bt_Susan* are the set of nodes $\{pc_Susan, mc_Susan, pc_Allen, mc_Allen, bt_Allen\}$. Note that there is an indirect

causal effect trail from pc_Allen to bt_Susan, via mc_Susan. Also, there is a common cause trail between bt_Allen and bt_Susan via $\{mc_Allen, mc_Susan\}$. Additionally, pc_Allen is able to influence the beliefs in mc_Allen via bt_Allen, as they form a common effect trail (v-structure). Influence cannot flow from bt_Brian and pc_Brian to bt_Susan, as they are blocked by the evidence of pc_Susan.

7.1.2 Inference in Bayesian Networks

Typically, a Bayesian network is used to compute marginal distributions of one or more query nodes, given some of the others nodes are clamped to observed values. Inference is the computation of these marginal probabilities, defined as

$$P(\mathbf{X}|\mathbf{E}) \propto \sum_{\mathbf{Y} \neq \mathbf{X}, \mathbf{E}} \mathbf{P}(\mathbf{X}, \mathbf{E}, \mathbf{Y}) \tag{7.3}$$

where $\mathbf{E}$ represents a set of observed variables (the evidence), $\mathbf{X}$ is the set of unobserved variables whose values we are interested in estimating, and $\mathbf{Y}$ are the variables whose values are missing (the hidden nodes). For very small Bayesian networks it is easy to marginalize sums directly. Unfortunately, the number of terms in the sum will grow exponentially with the number of missing values in a network, making the exact computation of marginal probabilities intractable for arbitrary Bayesian networks, In fact, this is known to be a NP-hard problem (Cooper, 1990; Dagum and Luby, 1993). The good news is that in some cases it is possible to exploit the graph structure, so that the exact computation of marginal probabilities has complexity linear in the size of the network. This is precisely the case of *polytree* networks. In polytrees, exact inference such as variable elimination (Li and D'Ambrosio, 1994) can be applied efficiently, by exploiting the chain-rule decomposition of the joint distribution and marginalizing out the irrelevant hidden nodes. There is also a family of algorithms named *junction tree*, which work by joining variables in cluster nodes so that the resulting network becomes a polytree.

7.1.3 Learning Bayesian Networks from Data

Typically, learning in Bayesian networks has as goal to return a model $B^* = (G^*, \theta^*)$, where G^* is a Directed Acyclic Graph and θ^* is a set of probability parameters. They

both together should precisely capture the underlying probability distribution P^* governing the domain. To do so, both the structure of the Bayesian network (the DAG) and local probability modes (CPDs) may be learned using an Independently and Identically Distributed (IID) data set $D = \mathbf{d}[\mathbf{1}], ..., \mathbf{d}[\mathbf{M}]$ of M examples sampled independently from P^*. However, the ideal goal is generally not achievable because of (1) computational costs and (2) limited data set providing only an approximation of the true distribution. Thus, learning algorithms attempt to return the best approximation to $B*$, according to some performance metric. As the metric is often a numerical criterion function, the learning task can be seen as an optimization problem, where the *hypothesis space* is the set of candidate models and the criterion for qualifying each candidate is the *objective function.* One common approach is to find B that maximizes the *likelihood* of the data, or more conveniently its logarithm, since the products are converted to summation. The Log-Likelihood (LL) is defined as

$$LL(B|D) = \sum_{k=1}^{M} \log P(\mathbf{d}[\mathbf{k}]) = \sum_{\mathbf{k=1}}^{\mathbf{M}} \sum_{\mathbf{j=1}}^{\mathbf{V}} \log \mathbf{P}(\mathbf{d}[\mathbf{k}]_{\mathbf{j}}|\mathbf{parents}(\mathbf{d}[\mathbf{k}]_{\mathbf{j}})) \tag{7.4}$$

where each example $\mathbf{d}[\mathbf{k}]$ is composed of a set of V random variables $d[k]_j$. Note that the likelihood function has the property of *decomposability*, where each variable is decomposed in a separate term. From a statistic point of view, the higher the likelihood, the better the Bayesian network is of representing $P*$. When the model is learned by maximizing the likelihood or a related function, we have a *generative* training, since the models is trained to *generate* all variables (Friedman et al., 1997). Alternatively, if it is known in advance that the model is going to be used to *predict* values to random variables $\mathbf{X}$ from $\mathbf{Y}$, the training may be done *discriminatively* (Allen and Greiner, 2000; Greiner and Zhou, 2002; Grossman and Domingos, 2004), where the goal is to get $\mathrm{P}(\mathbf{X}|\mathbf{Y})$ to be as close to the real distribution $P^*(\mathbf{X}|\mathbf{Y})$ as possible. In this case, the objective function typically used is the *Conditional Log-likelihood (CLL)*, computed as

$$CLL(B|D) = \sum_{k=1}^{M} \log P(d[k]_i|d[k]_1, ..., d[k]_{i-1}) \tag{7.5}$$

where $d[k]_i$ is the class variable and $d[k]_1, ..., d[k]_{i-1}$ are the others variables.

Notice that

$$LL(B|D) = CLL(B|D) + \sum_{k=1}^{M} \log P(d[k]_1, ..., d[k]_{i-1})$$

Because of that, maximizing CLL leads to better classifiers than when maximizing LL, since the contribution of $CLL(B|D)$ is likely to be swamped by the generally much larger (in absolute value) $\log P(d[k]_1, ..., d[k]_{i-1})$ term. In fact, (Friedman et al., 1997) shows that maximizing the CLL is equivalent to minimize the prediction error, since this metric takes into account the confidence of the prediction. However, different from LL, CLL does not decompose into a separate term for each variable, and as a result there is no known closed form for computing optimal parameters.

There are three situations concerning the observability of the dataset one may encounter when learning Bayesian networks (Koller and Friedman, 2009):

1. The dataset is *complete* or *fully observed* in such a way that each example has observed values for all the variables in the domain.

2. The dataset is *incomplete* or *partially observed*, i.e, some variables may not be observed in some examples.

3. The dataset has *hidden or latent variables*, whose value is never observed in any example. It may be the case that we are even unaware of some variables (we do not know they exist), since it is never observed in the data. Despite of it, they might play a central role to understand the domain.

In respect to what must be learned from data to result in a Bayesian network, we have two distinct cases :

1. The graph is known (although it is not necessarily the correct one) and it is only required that the parameters are learned from data;

2. Both the graph structure and parameters are unknown and it is necessary to learn them both from the dataset.

Parameters estimation

The problem of CPDs estimation for a Bayesian network is concerned with estimating the values of the best parameters θ of a fixed graph structure. It is particularly

important because, although estimating the graph structure is arguably easy for an expert of the domain, eliciting numbers is difficult for people and may be unreliable. Besides the graph structure, the random variables and the values they can take are assumed to be known. The problem is usually solved by finding the *Maximum Likelihood Estimator (MLE)*, which is the set of parameters with the highest likelihood. However, others measure functions can be used.

When the dataset is complete, learning CPDs by MLE decomposes into separate learning problems, one for each CPD in the Bayesian network, and maximum likelihood estimation reduces to frequency estimation. Thus, to calculate the probability of a variable X assume value x when its parents assume the set of values *parents* it is used the formula

$$P(X = x|\ Parents(X) = \mathbf{parents}) = \frac{\mathbf{N(X = x, Parents(X) = parents)}}{\mathbf{N(Parents(X) = parents)}} \tag{7.6}$$

where $N(X = x, \mathbf{Parents} = \mathbf{parents})$ is the number of training instances in the dataset where X has the value x and its parents have the values **parents**.

When the dataset is only partially observed, the maximum likelihood estimation cannot be written in a closed form, since it is a numerical optimization problem with all the known algorithms working under nonlinear optimization. Algorithm such as *Expectation-Maximization (EM)*(Dempster et al., 1977), (Lauritzen, 1995), (McLachlan and Krishnan, 1997) and gradient descent (Binder et al., 1997) are used in this case. Here we give an overview of the popular EM algorithm.

EM algorithm EM is based on the idea that since it is not possible to calculate the real frequency counting, because some of the examples may have missing values, such counting should be estimated from the data. Then, these estimates can be used to maximize the likelihood. The algorithm assumes an initial set of parameters, which may be initialized at random, and iteratively performs the two steps below until convergence:

E-step: Based on the dataset and the current parameters, the distributions over all possible completion of each partially observed instance are calculated (ex-

pected counts). In order to calculate such distributions, the procedure must infer the probability of each variable for each value of its domain, using the current CPDs.

M-step: Each completion estimated above is treated as a fully observed data case, weighted by its probability. Then, each (weighted) frequency counting is used to calculate the improved parameters. The log-likelihood is calculated and in case it converges, the procedure stops. Otherwise, it iterates to E-step, but now using the newly computed CPDS.

Structure Learning

Often it is easy for a domain expert to provide the structure of the network, since it represents basic causal knowledge. However, this is not always the case and sometimes the causal mode may be unavailable or subject to disagreement. In these situations, it is necessary to search for a graph fitting the dataset. Learning a whole Bayesian network gets more complicated in the presence of incomplete data, since inference is necessary to produce the expected counts, and the problem is compounded in the presence of hidden variables: in this case the domain and number of hidden variables must also be selected from dataset.

Using a naive approach, structure learning may start with a graph with no edges and iteratively add parents to each node, learning the parameters for each structure and measuring how good is the resulting model using an objective function. Or still, it may start with a random generated initial structure and use a search approach to modify the current structure, by adding, deleting or reversing edges. Since the resulting graph structure cannot have cycles, it is necessary to either define beforehand an ordering to the variables or search for possible orderings (Russell and Norvig, 2010). To measure how well a model explains the data, one may use a probabilistic function such as log-likelihood, perhaps penalizing complex structures by using for example the MDL principle (Lam and Bacchus, 1994). Is is important to note that for each graph candidate parameters may change and therefore they will need to be re-estimated. In case the dataset is complete, only the variables involved with the modification must have theirs CPDs re-learned (decomposability property). Unfortunately, this does not happen when the dataset is incomplete,

since local changes in the structure may result in global changes in the objective function and because of that the parameters of CPDs may change. In this case it is imperative to apply a heuristic solution such as Structural EM (Friedman, 1998) to return the final Bayesian network.

The situation gets even worse in the presence of hidden variables. Including hidden variables in a network can greatly simplify the structure by inducing a large number of dependencies over a subset of its variables. However, it is important to analyze the trade-offs, since learning a structure with hidden variables is far from trivial: it is necessary to decide how many of them are going to be included, their domains and also where to include them in the structure. If the number of variables and their domain is previously informed, it is possible to treat this problem as an extreme case of learning with incomplete data, where such variable(s) are never observed. There are several approaches developed to learning with hidden variables, typically based on the idea of using algebraic (Kearns and Mansour, 1998; Tian and Pearl, 2002) or structural (Elidan et al., 2000) signatures. Recently, we also contributed with a method for adding hidden variables in the structure of Bayesian networks, based on a discriminative approach and theory revision (Revoredo et al., 2009).

7.2 Bayesian Logic Programs: A Logical Probabilistic Model Extending Bayesian Networks

Although Bayesian networks are one of the most important efficient and elegant formalism for reasoning about uncertainty, they are a probabilistic extension of propositional logic (Langley, 1995). Indeed the qualitative component of a Bayesian network corresponds essentially to a propositional logic program. Consider, for example, the network exhibited in Figure 7.1. The influence relations of such a Bayesian network can be represented trough the propositional logic program in Table 7.1, where the random variables in the Bayesian network correspond to logical atoms and the direct influence relation corresponds to immediate consequence operator.

Bayesian networks thus inherit the limitations of propositional logic, namely the difficulties to represent objects and relations between them. In the example just

Table 7.1: Propositional Logic Program representing the network in Figure 7.1

bt_Susan	$\vert$	$mc_Susan, pc_Susan.$
bt_Allen	$\vert$	$mc_Allen, pc_Allen.$
bt_Brian	$\vert$	$mc_Brian, pc_Brian.$
bt_Lily	$\vert$	$mc_Lily, pc_Lily.$
bt_Gy	$\vert$	$mc_Gy, pc_Gy.$
bt_Anty	$\vert$	$mc_Anty, pc_Anty.$
mc_Susan	$\vert$	$mc_Allen, pc_Allen.$
pc_Susan	$\vert$	$mc_Brian, pc_Brian.$
mc_Allen	$\vert$	$mc_Gy, pc_Gy.$
pc_Allen	$\vert$	$mc_Anty, pc_Anty.$
mc_Lily	$\vert$	$mc_Gy, pc_Gy.$
pc_Lily	$\vert$	$mc_Anty, pc_Anty.$
$mc_Brian.$		
$pc_Brian.$		
$mc_Gy.$		
$pc_Gy.$		
$mc_Anty.$		
$pc_Anty.$		

discussed, a new family would require a whole new graph. However, if we had the set of definite clauses in Table 7.2, we can take advantage of the domain regularities, by applying the rules upon different variables binding.

Table 7.2: First-order Logic Program representing the regularities in the network of Figure 7.1

$bt(X)$	$\vert$	$mc(X), pc(X).$
$mc(X)$	$\vert$	$mother(Y, X), mc(Y), pc(Y).$
$pc(X)$	$\vert$	$father(Y, X), mc(Y), pc(Y)$

Bayesian Logic Program (BLP) (Kersting and De Raedt, 2001d; Kersting and De Raedt, 2001a; Kersting, 2006) were developed to combine the advantages of the elegant and efficient formalism of the Bayesian networks with the expressive powerful representation language provided by Logic Programming. The goal is therefore to eliminate the disadvantages of propositionality in Bayesian networks and the lack of reasoning under uncertainty within ILP. The key idea is to associate the clauses above to a CPD, that characterizes the probability of the head of the clause, given

the literals in the body. This is exhibited in Table 7.3.

Table 7.3: First-order Logic Program representing the regularities in the network of Figure 7.1, where each CPD is represented as a list of probability values.

$bt(X)$	$\mid mc(X), pc(X)$, [0.97, 0.01, 0.01, 0.01, ..., 0.01, 0.01, 0.01, 0.97].
$mc(X)$	$\mid mother(Y, X), mc(Y), pc(Y)$, [0.93, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, ..., 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.93].
$pc(X)$	$\mid father(Y, X), mc(Y), pc(Y)$, [0.93, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, ..., 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.93].
$mc(X)$,	[0.3, 0.3, 0.4]
$pc(X)$,	[0.4, 0.3, 0.3]

Next we define the BLPs language in more details.

7.2.1 Bayesian Logic Programs: Key concepts

Bayesian atoms and predicates A Bayesian predicate p/n is a first-order predicate with arity n and a a set of finite and mutually exclusive states associated to it (its domain). An atom of the form $p(t1, ..., tn)$ is a Bayesian atom if p/n is Bayesian, and both share the same domain. A Bayesian predicate p/l is generically a set of random variables, while a Bayesian ground atom $p\theta$ is a random variable over the states $domain(p/l)$.

Bayesian Clause A Bayesian clause c is an expression of the form $A|A_1, ..., A_n, n \geq 0$, where A is a Bayesian atom and each A_i is a Bayesian atom or a logical atom. The symbol | is used to highlight the conditional probability distribution. The set of Bayesian atoms in the body of c directly influence the Bayesian atom in the head of c. Thus, there is a CPD $cpd(c)$ associated with each Bayesian clause c, encoding $P(head(c)|body(c)$ and representing the conditional probabilities distributions of each ground instance $c\theta$ of the clause c. Logical atoms do not have probabilistic influence in the Bayesian atom in the head of c, Note that as logical atoms do not correspond to random variables they do not have a representation in $cpd(c)$, serving only to instantiate variables in the clause.

The clause $mc(X)|mother(Y,X), mc(Y), pc(Y)$ in the genetic example is a Bayesian clause, where $mc/1$ and $pc/1$ are Bayesian predicates with $domain_{mc/1} = domain_{pc/1} = \{a, b, 0\}$ and $mother(Y, X)$ is a logical atom, used to do the connection between logical variables X and Y. In this context, $mc(susan)$ represents the gene a person named $Susan$ inherits from her mother as a random variable over the states $\{a, b, 0\}$.

Combining rule It may be the case that several ground clauses have the same head, for example, there are two clauses c_1 and c_2 such that $head(c_1\theta_1) = head(c_2\theta_2)$. The clauses specify $cpd(c_1\theta_1)$ and $cpd(c_2)\theta_2)$ but not the required probability distribution $P(head(c_1\theta_1)|body(c_1\theta_1) \cup body(c_2\theta_2)$. Typically, in order to get such a distribution *combining rules* are employed. A combining rule is a function that maps finite sets of conditional probability distributions $\{P(A|A_{i1}, ..., A_{ini})|i = 1, ..., m\}$ onto one combined conditional probability distribution $P(A|B_1, .., B_k)$ with $\{B_1, ..., B_k\} \subseteq \cup_{i=1}^{m}\{A_{i1}, ..., A_{ini}\}$. For each Bayesian predicate p/l there is a corresponding combining rule $cr(p/l)$ such as *noisy-or* (Jensen, 2001), *noisy-max* (Díez and Galán, 2002), *(weighted)-mean* (Natarajan et al., 2008).

Definition 7.4 *A BLP consists of a set of Bayesian clauses. For each Bayesian clause c there is exactly one conditional probability distribution $cpd(c)$and for each Bayesian predicate p/l there is exactly one combining rule $cr(p/l)$.*

Well-defined BLP As a logical probability model, inference in BLP is executed in a Bayesian network generated from the BLP B and the least Herbrand model $LH(B)$ associated to the domain and the BLP. The nodes in the $DAG(B)$ correspond to ground atoms in $LH(B)$, encoding the direct influence relation over the random variables in $LH(B)$. Thus, there is an edge from a node X to a node Y if and only if there is a clause $c \in B$ and a substitution θ such that $y = head(c\theta)$, $x \in body(c\theta)$ and for all ground atoms $Z \in c\theta, z \in LH(B)$. Although indeed the Herbrand base $HB(B)$ is the set of all random variables we can talk about, only the atoms in the least Herbrand model constitute the relevant random variables, as they are the true atoms in the logical sense, and because of that only they are going to appear in the

$DAG(B)$. Each node in $DAG(B)$ is associated to a CPD $cpd(c\theta)$ where $head(c\theta) = y$, after applying a combining rule of the corresponding Bayesian predicate to the set of CPDs related to the corresponding variabilized clause. We say a BLP B is *well-defined* if and only if $LH(B) \neq \emptyset$, the graph associated to B is indeed a DAG and each node in $DAG(B)$ if influenced by a finite set of random variables. If B is well defined, it specifies a joint distribution $P(LH(B)) = \prod_{X \in LH(B)} P(X|parents(X))$ over the random variables in $LH(B)$.

7.2.2 Answering queries procedure

As well as in Bayesian networks, any probability distribution over a set of random variables can be computed in a BLP. A probabilistic query to a BLP B is an expression of the form

$$? - Q_1, ..., Q_m | E_1 = e_i, ..., E_p = e_p$$

where $m > 0$, $p \geq 0$, $Q_1, ..., Q_m$ are the query variables and $E_1, ..., E_p$ are the evidence variables and $\{Q_1, ..., Q_m, E_1, ..., E_p\} \subseteq HB(B)$. The answer for such query is the conditional probability distribution

$$P(Q_1, ..., Q_m | E_1 = e_i, ..., E_p = e_p)$$

The least Herbrand model and consequently its corresponding Bayesian network may become too large to perform inference. However, it is not necessary to compute the whole Bayesian network but instead only a part of it, called the *support network*. The support network N of a random variable $X \in LH(B)$ is the induced sub-network of $\{X\} \cup \{y|Y \in LH(B)\ and\ Y\ influences\ X\}$. The support network of a finite set $\{X_1..., X_n\} \in LH(B)$ is the union of the support networks for each variable X_i. Thus, the support network constructed to answer a probabilistic query consists of the union of the support networks for each query variable and each evidence variable. (Kersting and De Raedt, 2001b) proved that a support network of a finite set $\mathbf{X} \subseteq \mathbf{LH}(\mathbf{B})$ is sufficient to compute $P(\mathbf{X})$.

In order to construct the support network of a query variable X it is necessary to gather all ground clauses employed to prove $: -X$. The set of all proofs is all

the information needed to compute the support network. The proofs are typically constructed by using the SLD-resolution procedure. After that, it may be necessary to combine multiple copies of ground clauses with the same head using the correspondent combining rule. Finally, to indeed calculate the probability distribution, the support networks of each Bayesian atom which is part of the probabilistic query are united, giving rise to a DAG. Such a DAG, together with the CPDs resulting or not of combining rules, and the states of each evidence variable, can be provided to any inference algorithm so that the probability of the query is computed.

Algorithm 7.2 Algorithm for Inducing a Support Network from a Probabilistic Query (Kersting and De Raedt, 2001a)

Input: A probabilistic query $? - Q_1, ..., Q_n | Ev_1 = ev_1, ..., Ev_m = ev_m$
Output: A support network N related to the probabilistic query
1: **for** each variable $X_i \in \{Q_1, ..., Q_n, Ev_1, ..., Ev_m\}$ **do**
2: compute all proofs for X_i
3: extract the set S of ground clauses used to prove X_i;
4: combine multiple copies of ground clauses $H|B \in S$ with the same H, generating the support network N_i for X_i;
5: $N \leftarrow \cup_{i=1}^{k} N_i$;
6: $N \leftarrow prune(N)$;

Consider, for example, the logical part of a BLP in Table 7.4 and the query

$$?{-}bt(susan)|bt(brian) =' a', pc(brian) =' a', pc(susan) =' o', pc(allen) =' o', bt(allen) =' a'$$

Table 7.4: Qualitative part of a Bayesian Logic Program

$bt(X)\|\ mc(X), pc(X).$
$mc(X)\|\ mother(Y, X), mc(Y), pc(Y).$
$pc(X)\|\ father(Y, X), mc(Y), pc(Y).$
$mother(allen, susan).$
$father(brian, susan).$
$pc(allen).$
$mc(allen).$
$mc(brian).$
$pc(brian).$

Predicates *mother* and *father* are deterministic. In order to prove the query $bt(susan)$, the first clause in the table is instantiated with the substitution $X/susan$.

Then, the procedure tries to prove $mc(susan)$ and gets $mother(Y, susan), mc(Y), pc(Y)$. The variable Y is substituted by $allen$, because of ground atom $mother(allen, susan)$, and then the algorithm reaches the ground atoms $mc(allen), pc(allen)$. The same is done to prove $pc(susan)$. From this set of proofs, the support network in Figure 7.4(a) is built, by adding one node to each ground Bayesian atom in the proof and including an edge from an atom directly influencing another atom, following the Bayesian clauses. Next, following the same procedure, support networks are built for the evidence atoms, giving rise to the networks (b), (c), (d), (e) and (f) in Figure 7.4. Next step is to unite all those support networks, which is done by merging the random variables shared by more than one network. The final network is exhibited in Figure 7.5.

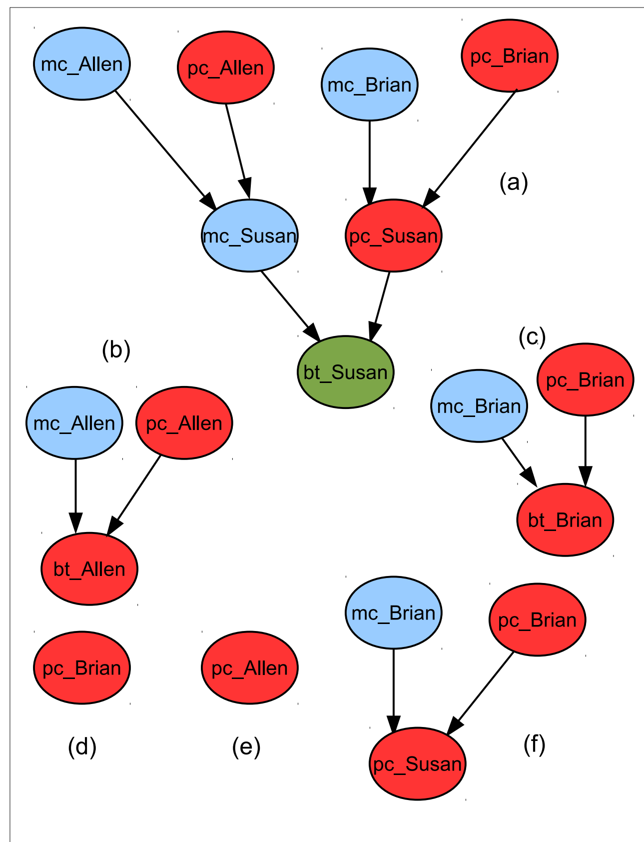

Figure 7.4: Support networks created from the query variable $bt(susan)$ and evidence variables $bt(brian)$, $pc(brian)$, $pc(susan)$, $pc(allen)$ and $bt(allen)$.

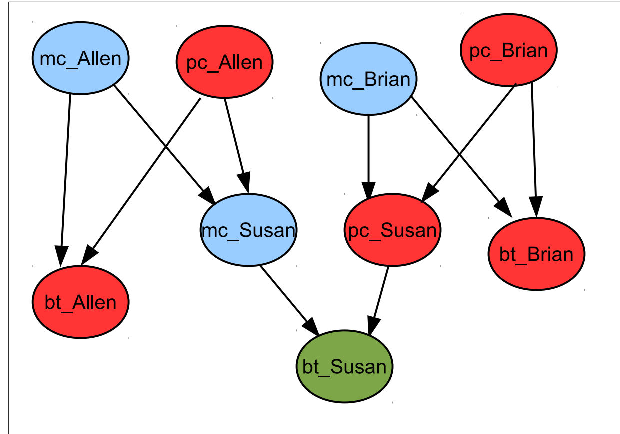

Figure 7.5: Bayesian network after joining same random variables from different support networks from Figure 7.4.

7.2.3 Learning BLPs

As usual in machine learning, BLPs are learned from a set of examples $\mathbf{D}$. Each example $\mathbf{D_l} \in \mathbf{D}$ has two parts, a *logical* part and a *probabilistic* part. The logical part is a *Herbrand interpretation*, more specifically the least Herbrand model of the BLP we want to learn. Thus, techniques used in *learning from interpretation* setting of ILP can be adapted for learning the qualitative part of the BLP, which implies that the hypothesis space is composed of sets H of Bayesian clauses such that all $\mathbf{D_l} \in \mathbf{D}$ is a model of H. Notice that, it is necessary to check whether the candidate hypothesis H is acyclic on the data, i.e, for each example of the data set the induced Bayesian network's graph must be acyclic. The probabilistic part is composed of a possibly partial assignment of values to the random variables in the least Herbrand model. One example of data case for the genetics example above would be:

$$\{bt(susan) =?, mc(susan) =?, pc(susan) =' o', pc(allen) =' o',$$
$$pc(brian) =' a', bt(brian) =' a', bt(allen) =' a', mc(allen) =?,$$
$$mc(brian) =?, mother(allen, susan), father(brian, susan)\}$$

where ? indicates a missing value for the random variable. Besides the logic part, a candidate hypothesis must also take into account the joint distribution over the random variables induced by the probabilistic part of the examples. To match

this requirement, the conditional probabilities distributions are learned from the data set and a probabilistic scoring function. Thus, the final goal is to find the hypothesis H, acyclic on the data, with the examples $D_l \in \mathbf{D}$ being models of H in the logical sense and where H best matches the examples according to the scoring function. To achieve this last requirement, the parameters of the associated conditional probability distributions in H must maximize the scoring function, which matches the learning setting of CPDs in Bayesian networks. Next, we see more details of how to estimate the parameters of BLPs and how to traverse the hypothesis space using refinement operators from ILP.

Parameter estimation in BLPs

In order to learn the parameters of a BLP, Bayesian networks are built from each example together with the current BLP. Thus, parameters estimation in BLPs follows the main ideas of parameters estimation in Bayesian networks. If the induced data set is completely observed, frequency estimation is achieved by counting. If there are missing values, it is necessary to use an algorithm capable of estimating the counts. However, parameters estimation differs from traditional algorithms of Bayesian networks because of the following reasons:

- In traditional Bayesian network setting, each node in the network has its separate CPT. In BLPs, CPDs are associated to Bayesian clauses rather than ground atoms what makes multiple instances of the same rule to share the same CPT. As a result, more than one node in the network may have the same CPT. Thus, parameters are learned considering Bayesian clauses instead of individual nodes in the network. Because of that, more than one node in the same network may be taken into account to learn the same CPT. This situation is illustrated in Figure 7.6, where nodes in magenta are yielded from the same Bayesian clause $c_i = bt(X)|mc(X), pc(X)$. Each one of them is going to be considered as a separate "experiment" to estimate parameters of the Bayesian clause that originates them. Thus, considering EM algorithm, parameters are estimated using the formula

$$cpd(c_i)_{jk} = \frac{\sum_{l=1}^{m} \sum_{\theta} P(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u_k}|\mathbf{D_l})}{\sum_{l=1}^{m} \sum_{\theta} P(body(c_i\theta) = \mathbf{u_k}|\mathbf{D_l})} \tag{7.7}$$

where u_j stands for the values in the domain of $head(c_i)$, u_k stands for each combination of values of the Bayesian atoms in $body(c_i)$ and θ denotes substitutions such that the example D_l is a model of $c_i\theta$. The formula represents the action of computing the CPD for a clause c_i by taking into account each node produced from a substitution θ applied to the clause. In Figure 7.6 substitutions θ applied to the Bayesian clause c_i are $\{X/susan\}, \{X/Allen\}, \{X/Brian\}$.

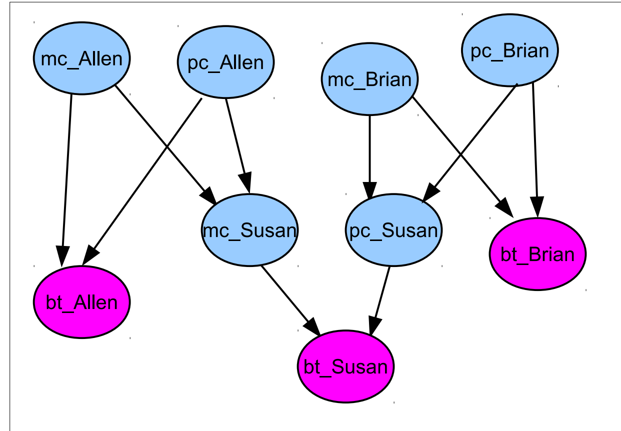

Figure 7.6: Bayesian network representing the blood type domain within a particular family. Nodes in magenta are yielded by the same Bayesian clause.

- The formula above computes the CPD of an *individual* clause c_i by estimating the joint probability of the head of the clause together with its body. The probabilities are computed through the Bayesian network produced by the examples. Thus, it is required that each node corresponds to a head of exactly one clause. In case combining rules are not handled properly, this is not going to happen with the Bayesian network produced from an example. Suppose, for example, that we have in a BLP the three clauses below:

  $pred(X)|pred1(X, Y)$.

  $pred(X)|pred2(X, Y)$.

  $pred(X)|pred3(X, Y)$.

  Then, suppose a substitution $\theta = \{X/1\}$. After the substitution, we have the ground clauses

$pred(1)|pred1(1,2)$.

$pred(1)|pred2(1,3)$.

$pred(1)|pred3(1,4)$.

The support network built from those ground clauses is going to be the one presented in Figure 7.7(a). Note that, as $pred(1)$ is the same random variable, the support network of each ground clause has been joined to produce the final network. The problem is, the requirement that each node is produced by exactly one clause is not attended if the support network is built like that. Therefore, it is necessary to assume independence of causal influences (ICI) (Heckerman and Breese, 1994), (Zhang and Poole, 1996), which allows that multiple causes on a target variable can be *decomposed* into several independent causes, whose effects are combined to yield the final value. As combining rules are employed to compute the final required probability distribution of the random variable, it is necessary to assume they are *decomposable*, i.e., they allow to decompose each random variable corresponding to the same ground head into several possible causes. This is expressed by adding extra nodes to the induced network, which are copies of the ground head atom. With this modification in the network, each node is produced by exactly one Bayesian clause c_i and each node derived from c_i (the yellow nodes) can be seen as a separate experiment for computing $CPD(c_i)$. Those extra nodes are considered hidden, since the value given in the example is relative to the instance itself and not to each representative of a rule. A network with the extra nodes is exhibited in Figure 7.7(b).

In this work, we follow the representation of combining rules defined in (Natarajan et al., 2008), where ground clauses are combined in two levels. The first level combines different instantiations of the same clause with a common head, while a second level combines different clauses with the same head. Suppose for example, in addition to the ground clause above, we would have also

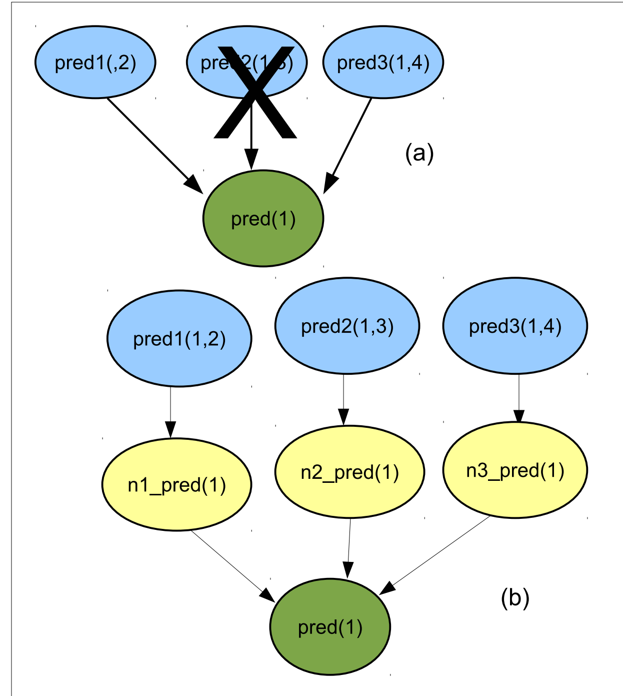

Figure 7.7: Figure (a) is a Bayesian network without adding extra nodes to represent decomposable combining rules. Node $pred(1)$ is produced by three different clauses. Figure (b) is the induced network reprsenting decomposable combining rules by adding extra nodes (yellow nodes) so that each node is produced by exactly one Bayesian clause. Nodes $n_?pred(1)$ has the domain of $pred$ and cpd(c) associated.

$pred(1)|pred1(1,3)$

$pred(1)|pred1(1,4)$

Then, instances from the first clause are combined in one level and the others clauses are combined in the second level. Different combination functions may be applied to each level. The final network is reproduced in 7.8.

Structure learning in BLPs

The algorithm proposed in (Kersting and De Raedt, 2001b) for learning BLPs traverse the hypothesis space using two ILP refinement operators: $\rho_s(H)$, which adds constant-free atoms to the body of a single clause $c \in H$ and $\rho_g(H)$, responsible for deleting atoms from the body of a single clause $c \in H$. The algorithm is reproduced

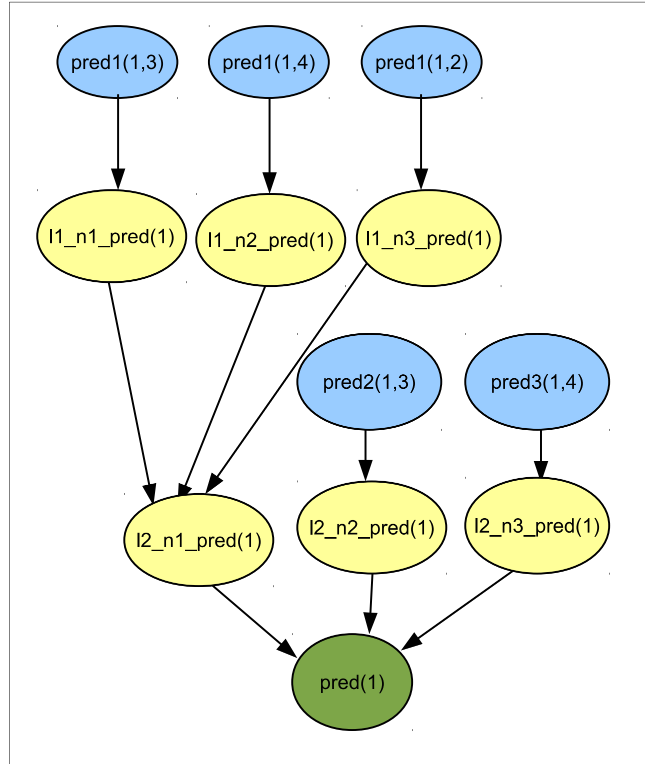

Figure 7.8: Decomposable combining rules expressed within a support network. Instantiations of the same clause and different clauses are combined separately. Nodes $l1_n?_pred(1)$ represent different instantiations of the same clause and nodes $l2_n?_pred(1)$ represent different clauses.

as Algorithm7.3 and work as follows. First, a hypothesis H_0, possibly generated from a learning from interpretation algorithm such as CLAUDIEN (De Raedt and Bruynooghe, 1993; De Raedt and Dehaspe, 1997), is assumed as the starting point and the parameters maximizing the log-likelihood $LL(D, H)$ are computed. Then, following a basic greedy hill-climbing, the neighbors of H_0 are induced using the dataset D and the refinement operators $\rho_s(H_0)$ and $\rho_g(H_0)$, requiring that they induce acyclic Bayesian networks and proves all examples in D. The candidate hypothesis are scored using the set of support networks and a probabilistic scoring function and the one with the best evaluation is considered as the next hypothesis if it has an evaluation better than the current hypothesis. This process continues until there are no further improvements in the scoring function.

Note that the refinement operators may be applied in all Bayesian clauses of

Algorithm 7.3 Algorithm for learning BLPs (Kersting and De Raedt, 2001b)

Input: A finite dataset D, a set of Bayesian clauses H_0
Output: A BLP B
1: $\Theta \leftarrow$ parameters maximizing $LL(D, H)$
2: $H \leftarrow H_0 \cup \Theta$
3: compute $Score_H$
4: **repeat**
5: **for** each $H' \in \rho\ _g(H) \cup \rho\ _s(H)$ **do**
6: **if** H' proves D **then**
7: **if** the Bayesian networks induced from H' and D are acyclic **then**
8: $\Theta' \leftarrow$ parameters maximizing $LL(D, H')$
9: compute $score_{H'}$
10: **if** $score_{H'} > score(H)$ **then**
11: $H \leftarrow H'$
12: $score_H \leftarrow score_{H'}$
13: $\Theta \leftarrow \Theta'$
14: **until** there are no candidate hypothesis H' scoring better than H
15: **return** H

the current hypothesis in order to generate their neighbors, which arguably causes the structure learning to be very expensive. Next section, we present the first theory revision system designed to revise logical probabilistic models such as BLPs.

7.3 PFORTE: Revising BLPs from Examples

Revising first-order logic theories aims to start from an initial theory and modify it in order to return a more accurate model compared to learning from scratch techniques. Similarly, one may have an initial logic probabilistic model with only some points preventing it of correctly reflecting the domain. Such a model could be obtained from the following sources:

- A domain expert elicited the clauses which are not necessarily reflecting the dataset;
- An ILP or PLL system learned the model considering an old data set and now there are new examples not necessarily reflected by the old model;
- An ILP or PLL system learned the model considering the current dataset, but it could still be improved by revision techniques. In case it was learn from an

ILP system, the uncertainty in the domain was not considered and the revision could improve the model.

Usually, it would be more valuable to search for those revision points and modify them, instead of discarding the original model or proposing modifications to all its points, as it is done in BLP structure learning algorithm. This approach should not only produce more accurate methods but also reduce the cost of searching in the space of clauses and parameters. In the probabilistic logic case, new questions arise, since the starting model is composed by two related parts: the clauses and the probabilistic parameters. Thus, in order to check whether the model should be revised one must take into account these two components when proposing modifications. Moreover, as the inference to answer queries is performed in the induced probabilistic graphical model, it is also through them that we should look for the problematic points. With this motivation, we developed the first probabilistic first-order revision system designed to revise Bayesian Logic Programs. It was named PFORTE since it was built upon FORTE system (Revoredo and Zaverucha, 2002; Paes, 2005; Paes et al., 2005b; Paes et al., 2005a; Paes et al., 2006a). This section reviews the PFORTE system as published in (Paes et al., 2006a).

7.3.1 PFORTE: Key concepts

Task definition

- **Given:** an incorrect initial logical probabilistic logic model, a consistent set of examples, background knowledge composed of Bayesian and/or logical clauses.
- **Find:** a revised logical probabilistic model that is complete considering the given examples and has the highest score given some metric.

In order to find this "minimally revised" and complete model, PFORTE works in two steps: the first component addresses theory incompleteness by using generalization operators only; the second component addresses classification problems using both generalization and specialization operators.

PFORTE terminology is defined as follows:

Logical probabilistic model: The logical probabilistic model is a BLP, composed of a a set of Bayesian clauses, as defined in 7.2.1.

In the Genetics domain, concepts might include the blood type, *bt(person).*

Definition 7.5 (PFORTE Example) *The schema of an example (D_i) is:*

$$D_i = \{Instances, Deterministic, Evidences\}$$

An instance *is an instantiation of a predicate, with an associated value from the domain of the corresponding Bayesian predicate;* Deterministic *are literals do not representing random variables and* Evidences *are Bayesian atoms with a (partial) assignment of values from their respective domains.*

We built this definition following BLPs, where the logical part (qualitative) is each ground atom in $Instances$, $Deterministic$ and $Evidences$ sets and the assignment of values in $Instances$ and $Evidences$ is the probabilistic part (quantitative). The dataset might be composed of several examples. Instances and Evidences in one single example are mutually dependent, while different examples are assumed as independent from each other. In the Genetics domain, one example could be the one represented in Table 7.5.

Table 7.5: Format of an example in PFORTE system

[
$[bt(susan) ='o', bt(brian) ='a', bt(allen) ='a']$
$[mother(allen, susan), father(brian, susan)]$
$[mc(susan) =?, pc(susan) ='o', pc(allen) ='o', pc(brian) ='a', mc(allen) =?,$
$mc(brian) =?]$
]

Note that from a logical point of view all the instances are assumed as positive, so that the final model must cover them, and the class from which the instance belongs is indicated by the value associated to such an instance.

Definition 7.6 (Completeness) *Given a set of examples D, where $D_i \in D$ and $D_i = \{Insts, Dets, Evs\}$ we say that a BLP B is* complete *considering these examples if and only if*

$$\forall D_i \in D, \forall I_j \in logical\ part\ of\ Insts:\ B \cup Dets \cup logical\ part\ of\ Evs \vdash I_j$$

Provability is established using standard SLD-resolution, taking the instance to be the initial goal.

Definition 7.7 (Classification) *Given an instance we say that this instance is correctly classified if the probability value computed to its real class is higher than a pre-defined threshold. For example, consider D_i exemplified above. We say that the instance $bt(susan)$ is correctly classified, if using an inference method in its support network, the probability value computed to class a is higher than a pre-defined threshold. Otherwise we say that this instance was* misclassified.

7.3.2 The PFORTE Algorithm

PFORTE is a greedy hill-climbing system composed of two steps. The first step focuses on generating a theory as complete as possible, by addressing failed examples. The second step has as goal to induce a theory as accurate as possible, by addressing misclassified examples. Both steps follow the key ideas below in a greedy hill-climbing search:

1. Identification of *revision points*. In the first step, those are the clauses failing to prove instances and they are called *logical revision points*. In the second step, those are the points responsible for the misclassification of instances, discovered through a probabilistic reasoning mechanism (named as *probabilistic revision points*.

2. Proposal of modifications to the revision points using *revision operators*. As all the instances are positive and the first step aims to make the hypothesis proves all the examples, only generalization operators are necessary. This differs from the second step, where both generalization and specialization operators are applied, in an attempt to make the final hypothesis more accurate.

3. Score the proposed revisions. Each proposed revision, even the ones contemplated in the first step, is scored by a probabilistic evaluation function such as log-likelihood or conditional log-likelihood, since the ultimate goal is to reflect the joint probability distribution over the random variables in the data set.

4. Choice of the revision to be implemented. Both steps choose the revision with the highest score to be implemented, in case its score is better than the score of the current hypothesis. The first step requires the coverage is improved as well, while the second step requires the accuracy is also improved.

5. Stop criteria. The first step stops when there are no further generalizations capable of improving the coverage. The Second step stops the whole procedure when there are no further revisions proposed to the revision points capable of improving the accuracy.

Algorithm 7.4 shows how the key ideas are implemented. The first step focuses on generating a complete hypothesis by only addressing failed examples. To do so, it revises the initial hypothesis iteratively, using a hill-climbing approach. Each iteration identifies the logical revision points, where a revision has the potential to improve the theory's coverage. A set of revisions generalizing the current hypothesis are proposed and scored on the training set. The best one is selected and implemented in case the current coverage is improved. This process continues until the first step cannot generate any revisions which improve example coverage. It is expected that the revised hypothesis will be complete considering the data set.

This complete BLP is the starting point to the second step of PFORTE. This step tries to improve classification by modifying the probabilistic revision points. Both generalization and specialization operators are applied on such points and as in the first step, the revisions are scored, the best one is chosen and implemented if it improves the accuracy. This greedy hill-climbing process proceeds until either no further revision improves the scoring function or the probabilistic accuracy cannot be improved.

Next will detail the main parts of PFORTE's algorithm: generation of revision points, generation of possible revisions to this revision points and how to score the proposed modifications.

Generating Revision Points

Revision points are places in a theory where errors may lie. PFORTE considered two types of revision points:

Algorithm 7.4 PFORTE Algorithm (Paes et al., 2006a)

Input: An initial logical probabilistic model B, the background knowledge BK, and a set of examples D
Output: A revised logical probabilistic model B

1: **repeat**
2: generate logical revision points;
3: **for** each logical revision point **do**
4: generate and score possible revisions;
5: update best revision according to the score and also improving coverage;
6: **if** best revision improves coverage **then**
7: implement best revision;
8: **until** no revision improves coverage
9: **repeat**
10: generate probabilistic revision points;
11: **for** each probabilistic revision point **do**
12: generate and score possible revisions;
13: update best revision found according to the scoring function and improving the probabilistic accuracy;
14: **if** best revision improves the probabilistic accuracy **then**
15: implement best revision;
16: **until** no revision improves score

- **Logical revision points.** These points are generated in the same way FORTE does for generalization revision points, by annotating places in the theory where proofs of instances fail. These are places where the theory may be generalized in order to become complete. We follow the annotation process originally proposed by Richards and Mooney (Richards and Mooney, 1995): each time there is a backtrack in the proof procedure, the antecedent in which the clause failed is kept; this antecedent is a *failure point*. In addition, other antecedents that may have contributed to this failure, perhaps by binding variables to incorrect values (the *contributing points*) are also selected. Both failure and contributing points are considered logical revision points.

- **Probabilistic revision points.** These points are generated from misclassified instances, by following a prediction criteria which takes into account the probability distribution and the dependency among the Bayesian atoms. Thus, to generate the probabilistic revision points, the support networks built from the examples are collected and used to perform inference over the instances,

now represented as query random variables. The value inferred in the network for each query variable is compared to the original value from the dataset and, in case they differ, the instance is considered as misclassified. The probabilistic revision points are then any clauses taking part in the network of the misclassified instance.

Algorithm 7.5 Algorithm for generating Probabilistic Revision Points in a Logical Probabilistic Model such as BLP

Input: A LPM B; Background knowledge BK; A set of examples $\mathbf{D}$
Output: A set of revision points RP
1: $RP \leftarrow \emptyset$
2: **for** each example $D_i \in \mathbf{D}$ **do**
3: $d \leftarrow$ Bayesian network built from B, BK, D_i
4: **for** each random variable v in D_i which is an instance (query Bayesian atom) $\in D_i$ **do**
5: $valueNode \leftarrow$ class of v inferred by a probabilistic inference engine
6: $valueExample \leftarrow$ class of the Bayesian atom $\in D_i$
7: **if** $\exists$ $valueExample$ and $valueExample \neq valueNode$ **then**
8: $GRP \leftarrow$ nodes in d
9: **for** each Bayesian clause $c \in B$ **do**
10: **if** $head(c)$ unifies with a random variable in GRP **then**
11: $RP \leftarrow RP \cup c$
12: **return** RP

Revision operators

In order to be able to repair arbitrarily incorrect theories, revision operators must be able to transform any theory in the language into another. Depending on the type of revision points different operators can be used.

- **Operators for logical revision points.** To propose modifications for logical revision points we use all FORTE generalization operators. The differences are in delete_antecedent and add_rule operator.

  a) *delete_antecedent* operator in FORTE deletes as many antecedents as possible while negative examples are not proved. As we do not have negative examples, this operator stops removing antecedents when examples became proved or the scoring function cannot be improved.

b) Original *add_ruleoperator* works in two steps after copying the incorrect clause. First, it removes the failed antecedents in order to prove previously unproved positive instances and then it adds new antecedents in an attempt to not cover negative examples. In PFORTE this operator was changed so that after removing the failed antecedents in a rule, PFORTE adds antecedents in an attempt to improve the value of the probabilistic scoring function.

- **Operators for probabilistic revision points.** To propose modifications for probabilistic revision points PFORTE also uses FORTE operators, in this case generalization and specialization operators, with slight modifications. When specializing clauses, PFORTE differs from FORTE in three different actions:

  1. Antecedents with the highest score are added while they are able to improve the score of the current hypothesis, differing from FORTE which stops to *addantecedents* when there are no provable negative examples and all the provable positive examples continue to be covered.
  2. FORTE may create more than one specialized version of the revision point in one revision, since a refinement of a clause can make provable positive examples become unprovable. As PFORTE cares for more than provability, it returns one specialized version of the original clause.

  Similarly, when deleting antecedents, the antecedent chosen to be removed is the one with the highest score. The algorithm for deletion/addition of antecedents is detailed in 7.6.

Algorithm 7.6 Algorithm for deletion/addition of antecedents in PFORTE

```
repeat
    for each antecedent in a clause/space of possible antecedents do
        if after deletion/addition covering of examples still holds then
            score this modification
    delete/add antecedent with the highest score
until no antecedent can be deleted/added without decreasing the score
```

Finally, the add_rule operator works combining both operators described above: first it deletes antecedents and then it adds antecedents. All operators for

probabilistic revision points must also enforce examples covering and range restriction of the clauses.

PFORTE_PI (Revoredo et al., 2006; Revoredo, 2009) was developed to incorporate two novel revision operators based on predicate invention (Muggleton and Buntine, 1988; Stahl, 1993; Muggleton, 1994; Kramer, 1995) in PFORTE system. The first operator, called *Compaction* creates new predicates to be head of clauses by absorbing a set of literals from the body of others clauses, in an attempt to decrease the complexity of literals and parameters in the LPM. The other operator, called *augmentation*, replaces a literal from the body of a clause by a new invented predicate, creates a clause with such a literal in the head and tries to specialize this new clause. As usual, the specialization and implementation of the predicate invention operators are only implemented if the score is improved.

Scoring possible revisions

Each proposed revision receives a score. Based on this score, PFORTE chooses the best one to be implemented. Since for each proposed modification the LPM is changed, it is necessary to re-learn the CPDs of the clauses. To do so, first PFORTE builds the support networks for each example generating a data set composed by these support networks. Having the set of support networks, the next step is to learn the CPDs for each clause, where *Maximum Likelihood Estimation*(MLE) approach is employed in the same way it is done in parameters estimation for BLPs. After learning and updating the CPDs for each clause, PFORTE calculates the score for the proposed modification using a probabilistic scoring function such as LL or CLL. The algorithm for scoring possible revision is detailed in 7.7

PFORTE was applied in three artificially generated datasets (Paes et al., 2005b; Paes et al., 2006a). The two-phase algorithm revising an initial BLP was successfully compared to the algorithm starting the learning from scratch (without an initial theory). However, when we tried to run PFORTE with more complex datasets, such as the ones used in SRL community, the system either could not finish the revision in reasonable time or it could not finish the revision at all, due

Algorithm 7.7 Algorithm to Score Possible Revisions in PFORTE

Input: A LPM B, a set of examples D, Background knowledge BK, a scoring function F

Output: Score $Score_B$ of B, considering D, BK and F

1: $S \leftarrow emptyset$
2: **for** each example $D_i \in D$ **do**
3: $S \leftarrow S \cup$ support network built for D_i
4: $\Theta^* \leftarrow$ learned CPDs considering S;
5: $score_B$ compute score using F and considering $Theta^*$, S;

to memory problems.

In (Mihalkova et al., 2007) it was developed an algorithm for transfer learning between two Markov Logic Network models, where the first step perform mapping between predicates and the second step revises the first model learned. The second step uses revision operators in a fashion similar to PFORTE.

Chapter 8

BFORTE: Addressing Bottlenecks of Bayesian Logic Programs Revision

Inference in Bayesian Logic Programs is performed on networks built from all the proofs of the set of examples. The algorithm for learning BLPs starts from a maximally general logic program satisfying the logical part of the examples and proposes refinements to each clause, by adding or deleting literals. In this fashion, learning BLPs requires searching over a large search space of candidate clauses, on the one hand, and the construction of all proofs to build the Bayesian networks, on the other. For each candidate theory one may have to perform full Bayesian inference, in order to compute scores in the presence of non-observed data.

The PFORTE system was developed to obtain accurate BLPs by revising an existing BLP and modifying it only in the rules used in Bayesian networks of misclassified examples. However, despite promising results on artificial domains, PFORTE faces similar bottlenecks as BLP and other SRL learning algorithms, as it must also address the large search space of logic programs and also perform inference.

The goal of this chapter is to address these bottlenecks of PFORTE system. First, the space of possible refinements can be reduced by limiting the candidate literals to be added to a clause to the ones present in the Bottom Clause (Muggleton, 1995). Second, we show that collecting revision points through Bayes Ball (Shachter, 1998) ultimately reduces the number of clauses marked as candidates for being revised by revision operators, as well as the type of the operator applied on them.

Third, we do not separate revision points as logical and probabilistic avoiding the cost of treating them in two separate steps of the revision process. Instead, the algorithm performs a single iterative procedure by considering all revision points as probabilistic ones. Fourth, we observe that in SRL example networks can overlap due to shared entities, resulting in large joint networks. We apply methods to (1) to overlap ground clauses with identical features, (2) to separate the requisite nodes for a specific queryand (3) to overlap requisite nodes found for different instances. Such methods follow the ideas of recent developments in lifted inference (Poole, 2003; Salvo Braz et al., 2005; Meert et al., 2010). By focusing on the relevant modification points and enabling the revision process to explore its full potentialities more efficiently, we expect to obtain an indeed effective and feasible revision system, as pointed out in (Dietterich et al., 2008) as a necessary development in SRL area. We named this new system BFORTE, referring to **B**LP, **B**ayes **B**all and **B**ottom clause.

The rest of this chapter is organized as follows. Section 8.1 address search space of new literals to the ones presented in the Bottom Clause. Section 8.2 address the search space of revision points by presenting the one step revision algorithm and the algorithm for collecting revision points with D-separation. Section 8.3 address the methods developed to reduce inference space, which is most often built to score each proposed revision. Section 8.4 shows experimental results obtained from BFORTE system and section 8.5 finally concludes the chapter.

8.1 Addressing Search Space of New Literals

The operators that generate the largest search space are the ones that search for new literals to be included in the body of clauses. This includes both add-antecedents and add-rule operators. PFORTE applies FOIL (Quinlan, 1990) top-down strategy to make new literals. As expected of a top-down approach, the number of generated literals may be huge, depending on their definitions and the background knowledge. Moreover, the literals generated will often not be much significative, as no insight is given on which terms should be an input/output variable or a constant. Whenever we add a new literal to a body of a clause, each generated literal must be scored so

that the best is chosen. Adding literals to Bayesian clauses causes a modification to the qualitative part of the BLP, so it is necessary to re-learn the parameters affected by the revision. Taking everything into consideration, adding literals to clauses outweighs the time expended proposing modifications and dominates the runtime of the revision process.

In BFORTE we use the Bottom Clause to compose the search space of evaluated literals to be added to a clause. The use of the Bottom Clause greatly decreases the number of possible literals, by limiting the candidate clauses to subsets of the Bottom clause. The use of the Bottom Clause in BFORTE follows the procedure devised in (Duboc et al., 2009) with small differences in the implementation. The most significant difference is in that the Bottom Clause is generated from a positive example, covered by the current clause, since their goal when specializing clauses is to continue proving positive examples while making negative examples become unprovable. Here, the Bottom Clause is generated from a misclassified instance, no matter its class. It is still required that the instance is covered by the base clause, since the clause before being refined is a subset of the bottom clause.

As in (Duboc et al., 2009) literals in the clause must obey mode declarations. Additionally, through determinations declarations, it is possible to define an ordering to random variables. Remember that a determination declaration $determination(pred_1/N1, pred_2/N2)$ indicates that $pred_2/N1$ may appear in the body of a clause whose head comes from $pred_1/N2$. By exploring all determinations, the system can identify which random variables can be a parent to another random variable. Those are random variables produced by literals in the place of $pred_2/N2$. Moreover, it can also identify the opposite: which random variables may to be children of another random variable in the Bayesian network of an example. Those are random variables generated from predicates in the place of $pred_1/N1$ in determinations declarations.

Bayesian networks may be used to perform two reasoning strategies: bottom-up reasoning, when it goes from effects to causes, and top-down reasoning, when the reasoning goes from causes to effects (Spirtes et al., 2001; Pearl, 2009). We can compare these reasoning strategies to the steps performed by the operator when

taking into account the Bottom Clause: first, the bottom clause is generated from the instance, which is similar to search for causes from an effect (the instance); next, the literals in the bottom clause may be added to the body of a Bayesian clause – the algorithm tries to improve the BLP by adding causes to effects. In fact, the causal relationship is used as a guide to construct the graph structure. Note, however, that the process of constructing the Bottom Clause and taking literals from it comprises only one level of reasoning, since only literals *directly influencing* the instance, i.e., the ones with an edge going to the head random variable, are collected. Thus, the literals in the Bottom Clause representing random variables may be direct causes of the effect represented by the random variable in the head of the Bayesian clause.

It is important to notice that the Bottom Clause procedure is instance driven. Literals are collected considering only one instance, making the choice of literals biased by such an instance. Another instance might point out more worthwhile literals, i.e., literals whose corresponding random variables would produce a stronger influence and/or over more instances. One alternative strategy would be to collect the Bottom Clause for more than one single instance within an example. Arguably, the space of candidate literals would be larger. Thus, one could consider to reduce the search space by measuring the volume of information flowing between the two random variables involved – the one from candidate literals and the one from the head of the clause (Cheng et al., 2002a). We intend to investigate this strategy in future work.

8.2 Addressing Selection of Revision Points

8.2.1 Bayesian Revision Point

The key step of the revision process is to identify revision points. BLP classification strategy is to build Bayesian networks, grounded from the set of examples and the current Bayesian clauses, and perform inference on them in order to compute probabilities for queries. Thereby, it is natural to use the same mechanism to find the points responsible for a misclassification. PFORTE considers two types of revision points: logical and probabilistic ones. In the first case, in the same fashion

as logical theory revision, clauses failing to prove an instance are marked to be generalized. Recall that in PFORTE every instance is positive in the logical sense, and its evidence value defines the class to which the instance belongs. If the example is unprovable, PFORTE could not compute a probability for it. In the present work we do not consider logical revision points separated from probabilistic revision points. Instead, in case the example is unprovable, we compute a probability for it by always including a *default node* in the Bayesian network, i.e., a node ground from a most general clause for the instance predicate. If such a default node gives a probability below a threshold to the correct class, then it will be marked as a revision point. Considering that we may now select revision points from the Bayesian network constructed to the example, we define *Bayesian revision points* as follows.

Definition 8.1 (Bayesian Revision Point) *Let B be the Bayesian network built from Bayesian clauses and an example D. In case an instance from D is misclassified after applying an inference engine in B, there is a maximum set S of nodes together with their respective CPDs in B relevant to assessing the belief in the instance. Bayesian revision points are the Bayesian clauses used to yield nodes in S.*

Following such a definition, BFORTE requires only one iterative step to fix the BLP: at each iteration Bayesian revision points are found and revision operators propose refinements on them. Note that default nodes are only introduced when there are unprovable instances. They can also lead to Bayesian revision points, but in this case, the only possible revision proposed to them is to create new rules. Algorithm 8.1 presents the top-level procedure of BFORTE, modified from Algorithm 7.4 by removing its first step (line 1 to line 8). As before, the algorithm follows an iterative hill climbing procedure, where at each iteration Bayesian revision points are collected at first, revisions are proposed to the revision points and their score are computed. The revision with the best score is chosen to be implemented, but only if it is capable of improving the current score of the BLP.

An observation about score computation is necessary here. As the proposed revisions are going to modify the qualitative part of the BLP, it is also necessary to reflect those modifications in the quantitative part of the model. Thus, at each proposed modification we re-learn the parameters involved in the modified structure.

BLP revision presented in this work has an inherent discriminative behavior: we are concerned with predictive inference and revision points are found out through misclassified instances. In this way, it is more appropriate to employ a discriminative evaluation function such as conditional log-likelihood, root mean squared error or precision-recall based functions. Additionally, the model would benefit from a discriminative training of the parameters. Because of that, we implemented the gradient descent learning of parameters to minimize the mean squared error, as initially designed in (Natarajan et al., 2008). The method is based on representing combining rules in two levels: the first one combines different instantiations of the same rule, where they all share the same ground head, and the second one combines different clauses with the same ground head, as explained in section 7.2.3.

Algorithm 8.1 BFORTE Top-Level Algorithm

Input: A BLP BP, the background knowledge BK, a set of examples D, a set OP of revision operators to be considered to revise clauses
Output: A revised BLP BP'

1: learn probability parameters of BP
2: **repeat**
3: generate Bayesian revision points;
4: **for** each Bayesian revision point **do**
5: **for** each revision operator $\in OP$ **do**
6: propose revision
7: score revision
8: update best score revision found;
9: **if** best revision improves the current score **then**
10: implement best revision;
11: **until** no revision improves score

8.2.2 Searching Bayesian Revision Points through D-Separation

Originally, PFORTE denoted as probabilistic revision points all clauses whose head took part in the Bayesian network where an instance had been misclassified. Although the set of candidate clauses to be modified is smaller than in Kersting's BLPs learning algorithm, since not all clauses appear in every network, depending upon the size of the network several Bayesian clauses are going to be marked to be modified by the revision operators. Howevver, not all clauses that are used to generate the network may be relevant to the classification of the query.

Algorithms for learning Bayesian networks are generally grouped into two categories: scoring based methods (Heckerman, 1996; Tian, 2000; Chickering, 2002), which uses a heuristic to construct the graph and a scoring function to evaluate it; and constraint-based methods (Verma and Pearl, 1990; Spirtes et al., 2001; Cheng et al., 2002b; Campos, 2006), relying on conditional independece tests to build the model. There is also a recent effort in getting the best of both worlds, by developing strategies combining these two approaches (Tsamardinos et al., 2006; Pellet and Elisseeff, 2008). In this work, we follow the second group of algorithms when identifying Bayesian revision points, by designing a procedure that analyses the dependency relationships of a misclassified instance. Accordingly, *active paths* are searched to identify the nodes in the network that exerts influence over misclassified instances. By a path we mean any consecutive sequence of edges, without regarding their directionalities. A path is active if it carries information or dependence from one node to another through this sequence of edges. Whenever there is an active path between two variables, we say they are *d-connected*. Formally, this concept is defined as follows.

Definition 8.2 (D-connection for directed graphs (Pearl, 1988)) *For disjoint sets of vertices,* $\mathbf{X}$*,* $\mathbf{Y}$ *and* $\mathbf{E}$*,* $\mathbf{X}$ *is d-connected to* $\mathbf{Y}$ *given* $\mathbf{E}$ *if and only if for some* $X \in \mathbf{X}$ *and* $Y \in \mathbf{Y}$*, there is an (acyclic) path* $\mathbf{U}$ *from* X*,* Y *such that:*

- $\mathbf{U}$ *is composed of a direct connection for* X *and* Y*:* $X \leftarrow Y$ *or* $X \rightarrow Y$*;*
- $\mathbf{U}$ *indicates an indirect causal/evidential effect:* $X \leftarrow W \leftarrow Y$ *or* $X \rightarrow W \leftarrow Y$*,* $W \notin \mathbf{E}$*;*
- $\mathbf{U}$ *represents a common cause:* $X \leftarrow W \rightarrow Y$*,* $W \notin \mathbf{E}$*;*
- $\mathbf{U}$ *represents a common effect (a v-structure):* $X \rightarrow W \leftarrow Y$*,* W *or any of its descendants* $\in \mathbf{E}$.

If no path between the set of nodes is active, the variables are *d-separated*. In this case, there is a variable in the undirected path blocking influence from one node to another. To detect the nodes d-connected to the misclassified node, we take

advantage of linear-time Bayes Ball (Algorithm 7.1), starting from the misclassified query node. Algorithm 8.2 presents the top-level procedure for collecting revision points, which is performed as line 3 of Algorithm 8.1. First, it is necessary to identify the misclassified instances in each example. To do so, each instance at each example has its marginal probability computed. Note that when computing a marginal probability for a specific query, others instances in the same example are considered as evidence, as it is usually done in collective inference (Jensen et al., 2004). In this case the algorithm proceeds to run Bayes Ball, so that the active paths leading to the misclassified nodes can be identified, and consequently the relevant nodes are selected. The set of relevant nodes are considered as the requisite nodes (observed nodes that had been visited and nodes marked in the top) and also the nodes marked in the bottom, i.e., all nodes that are not discovered as irrelevant by the Bayes Ball algorithm. Finally, the clauses corresponding to the relevant nodes are identified and marked to be revised.

Algorithm 8.2 Top-level Selection of Revision Points

Input: A value of threshold T, A set N of Bayesian networks, each corresponding to an example
Output: A set of Bayesian Clauses RP, marked as revision points

1: $RP \leftarrow \emptyset$
2: **for** each Bayesian network $N_i \in N$ **do**
3:     **for** each query instance $id \in N_i$ **do**
4:         compute the probability $P(id = class)$ using an inference engine, where $class$ is the value of id in the dataset
5:         **if** $P(id = class) < T$ **then**
6:             $Relevant \leftarrow$ nodes marked as relevant by Bayes Ball Algorithm
7:             $RP \leftarrow RP \cup$ clauses corresponding to nodes in $Relevant$ set

8.2.3 Analyzing the Benefits Brought by Revision Operators According to D-Separation

In this section we analyze the behavior of revision operators by considering whether the modifications they propose are capable of modifying the influence flowing in the Bayesian network. The question that arises is whether the revision operators are capable of changing existing influence that is likely to make the instance misclassified and/or to create new connections influencing such instances. Additionally, to

check whether the operator really brings benefits to the BLP, this last is scored after modified using Bayesian networks built from the set of examples. In this way, the modifications proposed on the BLP form a hybrid mechanism combining independence tests and scored heuristic search methods.

To analyze the modifications proposed by the operators, we focus attention to the directions that the ball is bounced from/to nodes in Bayes Ball algorithm, aiming to disregard modifications that cannot bring additional or modify current influence over the query node. Additionally, we must also consider whether the modification can change the set of Bayesian clauses instantiated by the example to build the Bayesian network, since this also can change the influence coming to a node. Remember from section 7.2.2 that a directed edge in included between a pair of nodes if there is a ground clause whose head is represented by the node where the edge arrives and whose body contains a representative of the node from where the edge leaves. We proceed by discussing each possible situation ocurring in nodes marked as relevant separately.

Non-observed node visited from a child (NOC node) A non-observed node visited from a child passes ball from the child to (1) its parents, so that additional indirect causal influence might be brought, and to (2) its children, that can also bring back further indirect effect influence, as visualized in Figure 8.1. Naturally, the visiting child is also likely to influence the visited node. The node representing the misclassified instance itself is such a kind of this node, since the visit starts from it as it had been visited from a child, according to Algorithm 7.1. In this case, the set of operators are going to behave as follows, as graphically represented in Figures 8.2, 8.3, 8.4 and 8.5.

- Delete rule operator. As the clause whose head represents such a NOC node is deleted, the flow of influence through this path is withdrawn. Thus, this indirect causal/effect influence to the misclassified instance brought by this node is removed from the network. While this is a radical approach, it can reduce the size of the BLP and consequently of the Bayesian networks.
- Add antecedent operator. A NOC node passes ball to its parents, in an attempt

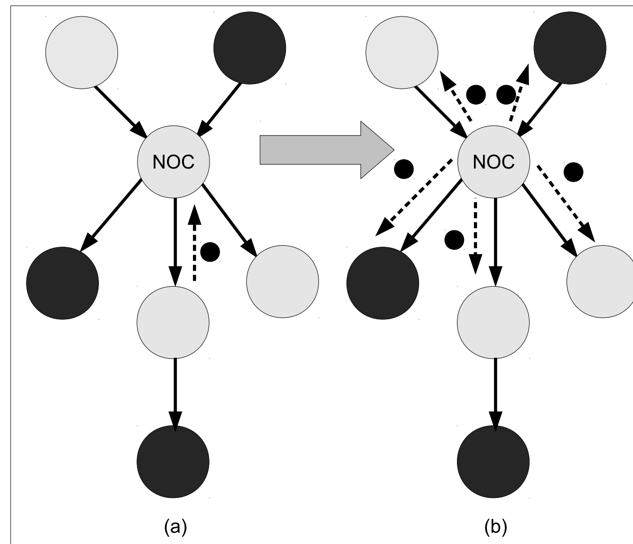

Figure 8.1: Non-observed node visited from a child, where shadow nodes are observed nodes. Figure (a) shows a non-observed node visited from a child. Figure (b) shows this node passing the received ball to its parents and children.

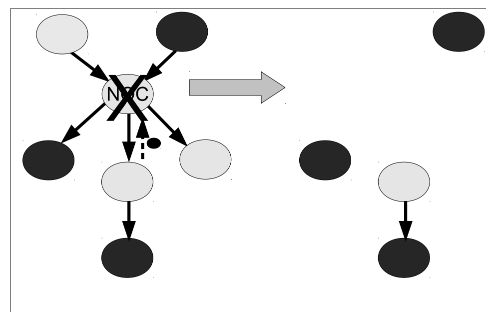

Figure 8.2: An example of the effect of deleting a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario.

to discover further influence to itself. Adding literals to the body of such a clause is going to add additional parents to clause head, and consequently a new flow of influence might be brought in, in the case of the examples remain provable after the modification. On the other hand, influence passing through this node is removed in the case of an example that becomes unprovable, similar to what happens in del rule operator. Note that this is different from a standard Bayesian operation, since here changing the set of clauses covering an

instance is also going to change the Bayesian network built from the example.

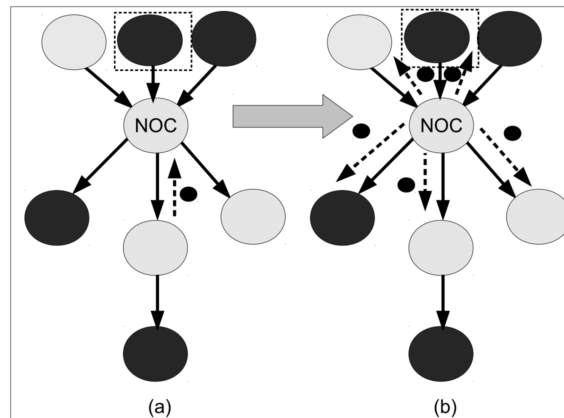

Figure 8.3: An example of the effect of adding antecedentes to a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario.

- Delete antecedents operator. In the same spirit to add antecedents might bring new profitable influence over the node, deleting antecedents may remove nodes whose influence through it from one of its parents contributes to wrong prediction.

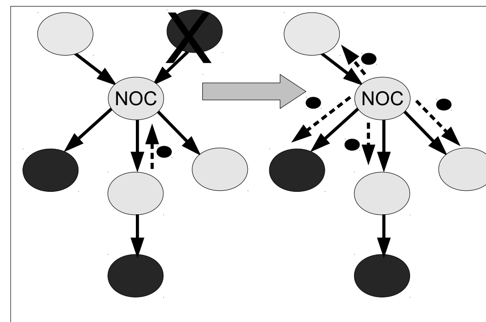

Figure 8.4: The effect of deleting antecedentes to a rule corresponding to a non-observed node visited from a child, in a misclassification node visiting scenario

- Add rule. As a new rule shares the set of satisfied examples with the old rule making rise to a NOC node, two situations may arise: (1) the new rule and old rule(s) give rise to the same ground atom in their head, which is going

to be included in the support network. In BLPs, this require that the final probability value is obtained through a combination function. The second possible situation (2) happens when the new and old rule do not share the same ground atom in their heads. In this case, the node coming from the head of the new rule may bring new influence, as it is likely to become a mate to the NOC node.

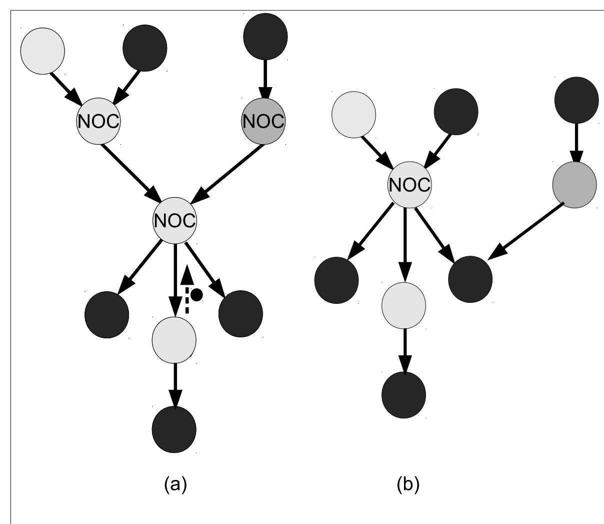

Figure 8.5: The effect of adding a new rule from a non-observed node visited from a child, in a misclassification node visiting scenario. Figure (a) shows a situation where the old and new rules had to be combined. Figure (b) shows the case where both rules were not combined.

Observed node visited from a child (OC node) Observe in Algorithm 7.1 that an observed node visited from a child blocks balls from children, i.e., if a ball comes from one of its children it is not passed anymore from the OC node. This happens because the evidence of the node makes the path from its children through it inactive. Therefore, an OC node d-separates its child of the others nodes in the graph, i.e., nodes which could be reached if it was possible to pass through an OC node. This is exhibited in Figure 8.6, where the arrows indicate the direction of the visit. Although OC nodes are marked as a revision point, since they are not considered as irrelevant for the misclassified instance, not all operators are going to propose profitable modifications to change/modify the influence exerted by them.

- In case the rule whose head corresponds to an OC node is *deleted* from the BLP, its child would lose its influence. As this last node is considered a requisite node, this operator may bring benefits to the model, in case the influence of such a node is bad to most of the query nodes. Therefore, this is an eligible operator to modify the set of influent connections to the misclassified instances.

- At first sight *adding antecedents* to the body of a clause whose head makes rise to an OC node, does not create new flow of influence, since any of its parents is blocked by its evidence. However, we must pay attention to the fact that those nodes are originated from Bayesian clauses, which are in essence logical clauses. In this way, add antecedents operator may remove some instances from the set of examples proved by the clause. The possible consequence is that the head of the clause is not able anymore to produce the OC node indicating a revision point. This is similar to the case of *delete rule* operator.

- Deleting antecedents from the body of a clause has two possible consequences in this case. First, the OC node is going to lose a parent. As its parents are blocked by its own evidence, this brings no changes in the flow of influence. Second, because a clause that has had one or mode antecedents deleted is more general than before, additional proof paths from the same clause may arise to build the Bayesian network. The ground heads of those additional paths either are the same as the previous OC node, or they induce new ground heads. In the first case, those equal nodes must be combined but besides sharing the same CPD (as they come from the same clause), they will also continue to contribute with the same evidence (as they come from the same ground fact). On the other hand, in case the modified clause produces new ground heads there is a chance that they are going to bring new influence to the child from whom the ball came. Consider, for example, a meaningless ground clause below that has given rise to an observed node in the network:

$$pred1(1,2) : -pred2(1,3), pred3(3,2), pred4(2).$$

Now, suppose $pred3(X,Y)$ is chosen to be deleted from the original clause.

One possible consequence would be the appearance of a new ground clause with the same ground head as before, say

$$pred1(1,2) : -pred2(1,_), pred4(2).$$

that had not appeared before because there is no $pred3(4,2)$ in the dataset. Note that both ground clauses have the same head and accordingly they are going to be combined. However, the random variable $pred(1,2)$, continues to be an observed node visited from a child, therefore, it is still a "blocking" node. On the other hand, it may be the case a new ground rule is produced after the deletion of the antecedent $pred3(X,Y)$, for example:

$$pred1(1,5) : -pred2(1,_), pred4(5).$$

Such a clause could not be produced before because there is no $pred3(_,5)$ in the background. This clause has a chance to bring new influence to the node from which the ball come, as $pred1(1,5)$ may become an ancestor of it.

- Add rule. Similarly to deleting antecedents case, adding new rules may bring additional evidence to the network.

Non-observed node visited from a parent (NOP node) Besides collecting evidence for itself, a non-observed node could be visited from its parent in an attempt to create an indirect effect path through one of its children, as can be seen in Figure 8.7. However, if this is really the case such a node would also be marked as non-observed visited from a child (NOC node) and therefore would attend a previous discussed case. Thus, deleting the rule corresponding to such a node head would only make the BLP, and consequently the Bayesian networks more compact. Similarly, adding antecedents to its body could only bring influence to their children, but this is also treated by the visit coming from a child case.

The operators left are the ones involving logic generalization. First, one may try to delete antecedents from such a clause in an attempt to generate new proof

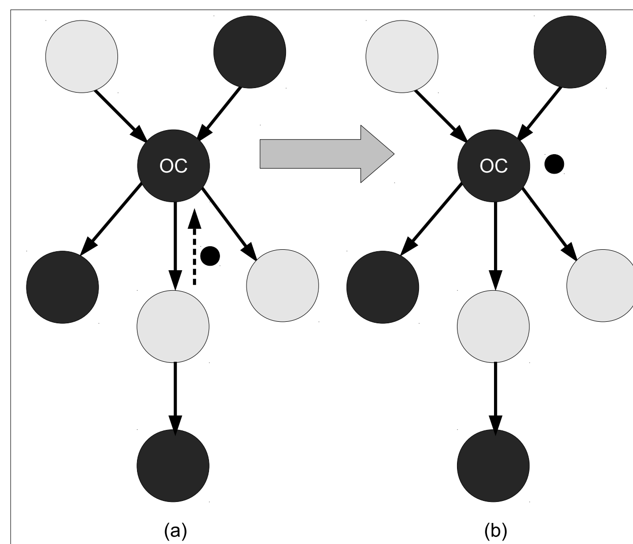

Figure 8.6: Observed node visited from a child, where shadow nodes are observed nodes. Figure (a) shows an observed node visited from a child. Figure (b) shows this node blocking the ball.

paths for some instance and consequently new ground heads forming a "brotherhood" relationship with the NOP node. In case the node visiting the NOP node remains as its parent (it is not one of the deleted antecedents), there will be a link to the node(s) coming from such new instantiations. There is a chance those new nodes bring together new evidence, as they are going to represent different ground facts of the dataset. The exact same situation may happen when adding new rules from such a NOP node. In this case, the link to the parent will exist if the visiting parent is also an element in the body of the just created clause.
Consider, for example the ground clause

$$pred(1,2)|pred1(1,3), pred2(3,4), pred3(4,2), pred4(2,2).$$

and assume the node yielded from $pred(1,2)$ is a NOP node, visited from the node produced from $pred1(1,3)$. Now, consider that the Bayesian predicate $pred3$ is deleted from the clause. This could generate a new ground clause, say

$$pred(1,4)|pred1(1,3), pred2(3,5), pred4(4,4)$$

since $pred(1,4)$ could not be proved by this clause before because there is no

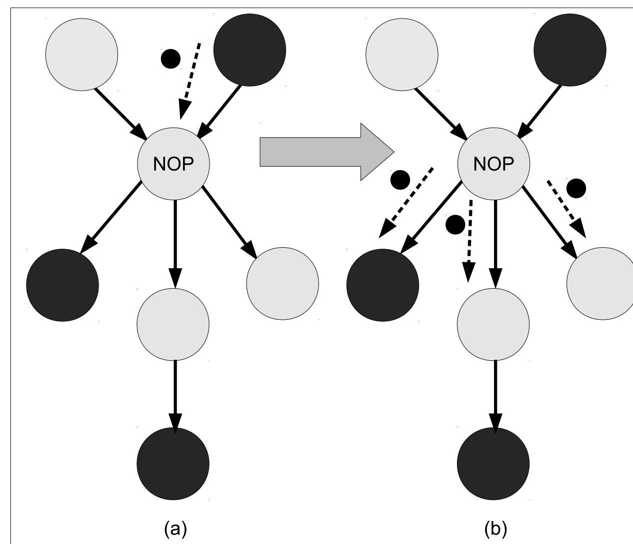

Figure 8.7: Non-observed node visited from a parent, where shadow nodes are observed nodes. Figure (a) shows a non-observed node visited from a parent. Figure (b) shows this node passing the ball to its children.

$pred3(5, 4)$ in the dataset. Note in Figure8.8 that there could be a new common cause trail in the network, due to this modification.

Observed node visited from a parent (OP node) The last case concerns an evidence node which is visited from a parent and bounces the ball back to its others parents, in an attempt to produce a common effect scenario, as the one presented in Figure 8.9. All operators may change the active paths as it is discussed next.

- Deleting the rule represented by such a node is going to cut it off the active path discovered by the visit. If the OP node contributes to a wrong prediction in this active path, removing the node from the network can repair misclassified instances.

- Deleting antecedents is beneficial for two reasons: firstly, as in previous cases, the generalization of the rule can bring additional active paths built from new instantiations of the rule, that did not exist before because of the antecedent(s) deleted. Secondly, deleting antecedents from the rule may remove the link to the visiting parent and, similar to the delete rule case, withdraw an influence

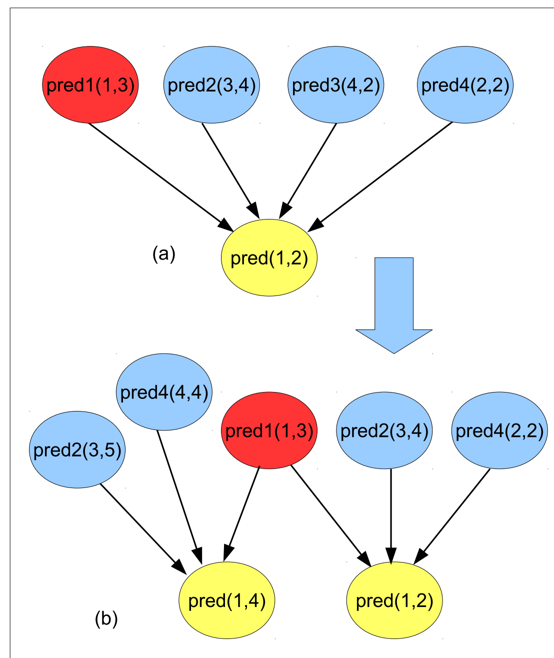

Figure 8.8: The effect of deleting antecedents from a clause whose head is a non-observerd node visited from a parent. Figure (a) is the network before deleting antecedents. Red node is the visiting parent and yellow node is the visited node. Figure (b) shows a possible resulting network, after the clause becomes more general.

contributing to the wrong prediction, exerted by the OP node. Moreover, deleting antecedents from such a node can remove common effect paths.

- Adding new rules has similar effects as the delete antecedents case: proofs not existing before may bring additional influence to the path, in case active path are also formed by the instantiations of the new rule and new common effect paths may arise from the added rule.

- Adding antecedents may bring two different consequences: the first one is similar to the result of deleting the rule, and may remove the link to the visiting parent, by making the clause more specific after some antecedents are added to it. Thus, the previous proof conducting to the parent node would fail. The second consequence is that adding antecedents in the body of such a clause might create further common effect influence path, which could be useful in correcting the misclassification of the example because of new evidence.

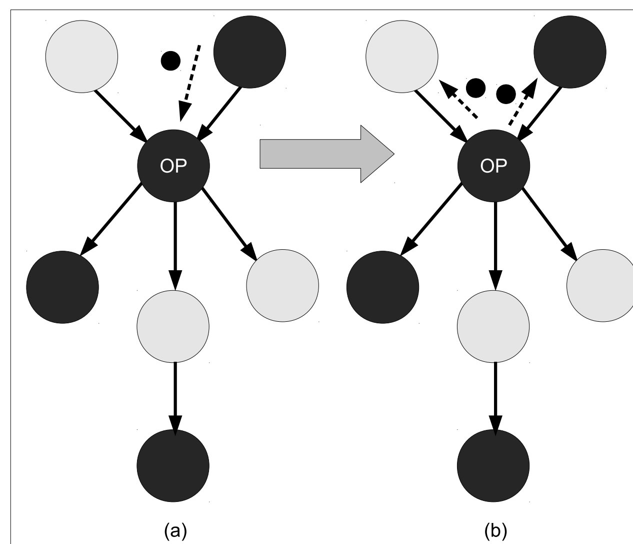

Figure 8.9: Observed node visited from a parent, where shadow nodes are observed nodes. Figure (a) shows a observed node visited from a parent. Figure (b) shows this node passing the ball back to its parents.

Remarks about Revision Operators and D-connection

One can see that are several issues concerning the real benefits a revision operator might bring. For example, some of the modifications are only likely to create/remove a connection when the set of ground Bayesian clauses changes after the revision. Moreover, some modifications are likely to bring/remove influence to fix misclassification, but this is not guaranteed, as they can/cannot make new active paths depending on the evidence. Moreover, some modifications depend whether a literal continues to be in the body of a clause, making sure a connection in the network is not removed. This can be true for one instance, but may not happen in other instances. As to check all possible cases is expensive, encompassing not only connections that are going to be removed but also of the active paths that are going to be created, we only disregard the operators guaranteed to not change influence that could fix the misclassified instance. Anyway, by computing the score of the revision it is verified whether the modification is going to improve the current BLP.

Notice that the revision operators used in this work do not compose a complete set of possible modifications to the BLP. To give an example, note that new active paths could be created by adding rules or antecedents leading a parent to visit

and/or be visited from a child. To achieve this goal, the revision would be able to mark revision points not only throughout the Bayesian networks but also from the knowledge base itself, by identifying all possible determinations leading to new connections coming from children to misclassified instances in the graph. Suppose, for example, that we have the Bayesian clause below, marked as revision point.

$$pred1(A, B) : -pred2(A, B).$$

It would be possible that a non existing clause could provide new information to $pred1/2$, say a clause such as

$$pred3(A, B) : -pred1(A, B).$$

As $pred3/2$ does not represent any instance in the examples and there is not an existing connection between pred3 and pred1, such a clause would not be proposed by the revision system.

BFORTE and its antecessor PFORTE cannot identify those possible new connections, as Bayesian revision points are identified through paths existing at the moment that an instance is misclassified. In this way, we cannot guarantee that we implement the complete set of possible modifications to improve classification in a BLP.

Finally, the BLP could be compacted by identifying clauses that are completely useless to any example in the dataset. Those correspond to nodes identified as irrelevant by Bayes Ball algorithm. To achieve this, Bayes Ball would have to run for each instance in the dataset. Then, the intersection of irrelevant nodes would correspond the clauses that could be safely removed from the BLP.

Algorithm 8.3 exhibits the procedure for applying revision operators to revision points, extended from lines 3-7 of Algorithm 8.1. It brings an optimization on the use of the four revision operators. First, when looking for revision points, it is necessary to keep the type each node matches, according to the four groups listed before. As the marked node is mapped to a Bayesian revision point (a Bayesian clause), the fact it is observed or not, visited from a child or from a parent, is stored together with the revision point. From our analysis, only deleting rules and adding

antecedents to a non-observed evidence are guaranteed to not present any chance to change the connections to the misclassified instance. In all other cases, the proposed revision may change the Bayesian network. Therefore, before proposing the revision, it is necessary to be sure that a Bayesian revision point is not identified only through that kind of visit.

Algorithm 8.3 Generation of Revision Points and Revisions Top-Level Procedure

Input: Set of Bayesian networks BN built from the dataset and current BLP
Output: Set of revisions Proposed to Bayesian revision points

1: $BRP \leftarrow$ Bayesian revision points identified through Algorithm 8.2, slightly modified to keep around the type of the node (OC, NOC, OP, NOP)
2: **for** each Bayesian revision point $BR \in BRP$ **do**
3:     **if** type of BR is not only NOP **then**
4:         propose modification to BR using all four operators
5:     **else**
6:         propose modifications to BR using delete antecedents and/or add rules operator
7:     compute the score for each proposed revision

Next section we discuss how to reduce the amount of the time expended in the revision by reducing the inference space.

8.3 Addressing Inference Space

In order to build Bayesian networks from the set of examples, PFORTE collects *all* ground clauses taking part in every possible way of proving instances and evidences in an example. Those ground Bayesian atoms become nodes in the Bayes net, and they are connected when there is a direct influence between them. As the instances and evidences in a single example are not i.i.d, each Bayesian network may have a large number of nodes and edges connecting evidences, instances and non-evidence nodes, leading to high inference run times.

Aiming to reduce the space where inference is performed, it is necessary to diminish the size of nodes to be considered by the inference engine. Making inference more efficient in relational probabilistic systems has been pursued by the development of lifted inference methods (Poole, 2003; Singla and Domingos, 2008; Milch et al., 2008; Meert et al., 2010), which mostly reduces the networks by exploring symmetries and grouping variables with identical behavior. Here, we decrease the

search space of the inference procedure by disregarding non requisite nodes of an instance. Non requisite nodes can be deleted from the network and we would still be able to resolve the query, as they are unable of influencing the instance. In addition, clauses and networks exhibiting identical behavior are grouped together so that they dot not have its score and parameters computed repeatedly unnecessarily.

8.3.1 Collecting the Requisite Nodes by d-separation

As said before, the networks built from examples are usually very large, since they are composed of the union of all possible proofs of each instance and each evidence. However, it is usually the case that for computing marginal probabilities for one instance only a subset of that large network is really relevant. We call the network composed of this subset of nodes as *minimum requisite network* for a query.

Definition 8.3 *The minimum requisite network g for a query node n is a subset of the original large network G built to the whole example, that contains only the requisite nodes necessary to compute the marginal distribution of n. All the others nodes in G can be safely disregard so that the query is still resolved by g.*

We make use of Bayes Ball algorithm (Shachter, 1998) to find out which nodes in the network are required to compose the minimum requisite network. In this way, instead of performing inference in one large network, the inference takes into account only a small subset of that network, namely the nodes which are really relevant for resolving the query. To collect such requisite nodes, the algorithm starts from one instance *id* in the original example and uses Bayes Ball to identify the minimum set of nodes required to compute the probability of *id*. This set is composed of visited observed nodes and those nodes marked in the top. Following collective inference (Jensen et al., 2004), the other queries instances in the example are treated as evidences when collecting the requisite nodes for *id*. Algorithm 8.4 brings the top-level procedure to identify requisite nodes and perform inference only over them.

Algorithm 8.4 is called at each time it is necessary to compute marginal probabilities, namely, to learn parameters and compute scores, when changes occur in the structure of the BLP.

Algorithm 8.4 Top-level Procedure for Performing Inference over Requisite Nodes Only

1: **for** each example $D_i \in D$ **do**
2: build the graph GD_i for D_i, joining support networks built from proof tree for each instance and evidence
3: **for** each instance $id \in D_i$ which is a node nid in GD_i **do**
4: consider nid as a query node
5: visit nodes in GD_i starting from nid, using Bayes Ball algorithm
6: $requisite \leftarrow$ visited observed nodes $\cup$ nodes marked in the top
7: compute probability distribtion for id using only $requisite$

Bayes ball was also used in SRL context to build up a lifted inference engine in (Meert et al., 2010). There, they apply the Bayes Ball directly in the ground graph construction procedure. We are going to investigate in the future whether that algorithm can save further time to the inference process.

8.3.2 Grouping together Ground Clauses and Networks

When constructing Bayesian networks relative to examples, every successful proof of an Bayesian atom is gathered together to build the graph of the net. However, it is often the case that a large number of those ground clauses shares exactly the same features: they are instantiated from the same Bayesian clause, and therefore they have the same conditional probability distribution; additionally, they have the same evidence associated to each of its Bayesian ground atom. In this way, the only difference among them would be the terms of ground Bayesian atoms. However, for the Bayesian network, atoms are mapped to nodes, which cannot "see" those terms anyway. Therefore, it is a waste of resources to represent each Bayesian atom obeying these conditions as different nodes in the graph. Consider, for example, the following meaningless ground clauses.

$$pred(A, B) : -pred1(C, A), pred1(C, B).$$

and suppose that for a ground atom $t1, t2$ we have several possible instantiations for the variable C, producing, for instance

$pred(t1, t2) : -pred1(r1, t1), pred1(r1, t2).$
$pred(t1, t2) : -pred1(r2, t1), pred1(r3, t2).$
$pred(t1, t2) : -pred1(r3, t1), pred1(r3, t2).$

...

$pred(t1, t2) : -pred1(r30, t1), pred1(r30, t2).$

Now, suppose that, say 25, of these share the same evidence for $pred1(X, t1)$, $pred1(X, t2)$. Instead of mapping each one of those nodes above to a separate node in the Bayesian network, we could put together the $25X2$ nodes of the same evidence and $5X2$ separate nodes.

Thus, to prevent this waste of resources performed by BLP network construction, we developed a procedure for grouping together different instantiations of the same Bayesian clause that behaves in exactly the same way. This group of similar nodes make rise to a *mega node* in the network, instead of only a single node.

Definition 8.4 *Let SN be the set of nodes originated from different successful proof paths created using Bayesian clauses Bcc and an instance ins. A mega node is defined as the grouping of a set of nodes in SN, which are relative to the same Bcc, and that shares the same evidence.*

Because of that, a random variable in the graph is not necessarily a single ground Bayesian atom, but could be representative of a group of several Bayesian atoms. The number of atoms mapped to the same node is kept so that it is taken into account when learning parameters. Notice that we employed the same schema of combining rules as (Natarajan et al., 2008), where instantiations of clauses with same head are combined in two levels: one level considers different instantiations of the same rule, while another level considers different rules with the same ground head, as explained in section 7.2.3. Thus, the clauses we mapped here to the same nodes would be combined anyway in the first level we just mentioned. However, although they would contribute in the same way for the final probability, since they share the same association of evidence values and CPDs, the inference engine could not have knowledge on that and it would compute their probability over and over again. Moreover, as decomposable combining rules are applied in BLPs, each different instantiation would be seen as a "separate experiment", by adding extra nodes in the network to represent them. By keeping the amount of ground atoms

overlapped together in a mega node, those nodes are still contributing as a separate experiment for the computation of parameters.

We also employ another way of overlapping components so that inference engines have to deal with a smaller space. This is the case when more than one graph have the same properties: the nodes are coming exactly from the same clauses, and evidence for those nodes are also the same. This is naturally a situation less frequent than the one we discussed immediately before, but still it can also happens mainly when Bayes Ball is applied to split the original networks in groups of smaller ones. To map more than one network to the same, it is required that (1) they have the same number of nodes; (2) they have the same edges connecting each pair of nodes (3) each group of child + parents are relative to the same Bayesian clause, so that the probability distribution associated to them in the different graphs are the same, and (4) the nodes have the same value of evidence, in case they are observed. The instances relative to networks grouped to one overlapped *mega network* are kept around, so that it can be taken into account when learning parameters or computing probabilities.

We show in Algorithms 8.5 and 8.6 the procedures for detecting and grouping together groups of nodes with the same behavior.

Algorithm 8.5 Top-level Procedure for Detecting Similar Ground Clauses and Mapping them to Only One

1: **for** each ground clause $clause_i\theta 1 \in$ proof trees of the same example **do**
2:     **for** each ground clause $clause_j\theta 2, i \neq j \in$ proof trees of examples **do**
3:         **if** $clause_i$ comes from the same Bayesian clause as $clause_j$ **then**
4:             **for** each Bayesian atom $a_{ki} \in clause_i\theta 1$ and $a_{kj} \in clause_j\theta 2$ **do**
5:                 **if** value associated to $a_{ki} \neq$ value associated to a_{kj} **then**
6:                     go to line 2
7:             remove $clause_j\theta 2$ from proof tree
8:             increase number of mapped clauses associated to $clause_i\theta 1$

8.4 Experimental results

This section we experiment our revision system BFORTE, by comparing each contribution developed in this chapter with a version of the system without the contribution. As there is no current implementation of BLP structure learning algorithm,

Algorithm 8.6 Top-level Procedure for Detecting Similar Networks and Mapping them to Only One

```
1: for each network N_i do
2:     for each network N_j, i ≠ j do
3:         if DAG(N_i) == DAG(N_j) then
4:             for each node n_ki ∈ N_i and node n_kj ∈ N_j do
5:                 if n_ki comes from a different clause as n_kj or n_ki has an evi-
                   dence value different from n_kj then
6:                     go to line 2
7:             increase number of group of nets associated to N_i
8:             associate instance(s) of N_j to corresponding node(s) of N_i
9:             disregard N_j
```

we use our own system to make the due comparisons. We opted by showing each contribution separately, so that it is possible to know the benefits brought by them.

Datasets We have considered datasets used in ILP and SRL communities. They are briefly described as follows.

1. UW-CSE is a vastly used dataset in SRL community (Singla and Domingos, 2005; Richardson and Domingos, 2006; Kok and Domingos, 2007). It consists of information about the University of Washington Department of Computer Science and Engineering. There are 5 examples, where each one of them contains instances representing a relationship of *advisedby* for a different research line of the department. There are 113 instances of the positive class and 2711 instances of the negative class, and 2673 ground facts.

2. Metabolism is based on the data provided by KDD Cup 2001 (Cheng et al., 2002b). The data consists of 6910 ground facts about 115 positive instances and 115 negative instances. As part of the facts represents the interaction between genes, we gathered together in one examples all the positive and negative instances.

3. Carcinogenesis is a well-known domain for predicting structure-activity relationship (SAR) about activity in rodent bioassays (Srinivasan et al., 1997). There are 162 instances from the positive class and 136 instances from the negative class and 24342 ground facts about them. This dataset is totally

separable, as each example is composed of only one instance.

Experimental Methodology The datasets were splitted up into 5 disjoint folds sets to use a K-fold stratified cross validation approach. Each fold keeps the rate of original distribution of positive and negative examples (Kohavi, 1995). To avoid overfitting during the revision process, similar to (Baião et al., 2003), we applied 5-fold stratified cross validation approach to split the input data into disjoint training and test sets and, within that, a 2-fold stratified cross-validation approach to split training data into disjoint training and tuning sets. The revision algorithm monitors the error with respect to the tuning set after each revision, always keeping around a copy of the theory with the best tuning set score, and the saved "best-tuning-set-score" theory is applied to the test set. The significance test used was corrected paired t-test (Nadeau and Bengio, 2003), with $p < 0.05$. As stated by (Nadeau and Bengio, 2003), corrected t-test takes into account the variability due to the choice of training set and not only that due to the test examples, which could lead to gross underestimation of the variance of the cross validation estimator and to the wrong conclusion that the new algorithm is significantly better when it is not.

The initial theories for Carcinogenesis and Metabolism were obtained from Aleph system using default parameters, except for clause length, which is defined as 5, noise, defined as 30, and minpos, set to 2. As UW-CSE is a highly unbalanced data, we use m-estimate as evaluation function and noise set for 1000. To generate such theories, the whole dataset was considered but using a 5-fold cross validation procedure. Thus, a different theory was generated for each fold and each one of these theories is revised considering its respective fold (the same fold is used to generate and revise the theories).

All the experiments were run on Yap Prolog (Santos Costa, 2008) and Matlab. To handle Bayesian networks, we have re-used Bayes Net Toolbox (Murphy, 2001), properly modified to tackle the particularities of BLPs. Combining rules are represented in two levels, where the first level (different instantiations of the same clause) uses *mean* as combining rule and the second level (different clauses with the same head) uses $weighted-mean$ as combination function. To learn parameters, we

implemented discriminative gradient descent described in (Natarajan et al., 2008) that minimizes mean squared error. For the UW-CSE, we use a weighted MSE, with the weight inversely proportional to the distribution of the classes, so that the parameters do not totally favor negative class because of its amount of examples. To perform inference, we adapted variable elimination inference engine to handle combination rules. We impose a minimum number of misclassified examples as 2 to a clause be considered as revision point, in order to avoid outliers. Threshold for indicating if an instance is correctly predicted is defined as 0.5.

8.4.1 Comparing BFORTE to ILP and FOL Theory revision

First of all, we would like to know whether BFORTE achieves better score results than standard first-order logic systems. We compared BFORTE after learning the initial parameters (after line 1 of Algorithm 8.1) and BFORTE after revising structure to Aleph and FORTE. The same theories Aleph learn are provided to both revision systems. The first two columns of Table 8.1 present the results of conditional log-likelihood achieved after learning initial parameters and after the revision process is finished, respectively. Third and fourth column present the score of BFORTE after learning initial parameters and after the revision process is finished. The last two columns presents the score of Aleph and FORTE, respectively. Bold faces indicate the best score results obtained from all systems. The symbol $\diamondsuit$ indicates the cases where score of BFORTE is significantly better than score of Aleph. The symbol $\bullet$ indicates the cases where BFORTE is significantly better than FORTE. Finally, $\star$ emphasizes the cases where it was possible to obtain an improvement after revising the structure.

Table 8.1: BFORTE compared to Aleph and FORTE

System/ Dataset	BFORTE Params CLL	BFORTE Struct CLL	BFORTE Params Score	BFORTE Struct Score	Aleph Score	FORTE Score
UW-CSE	-0.2142	-0.0746$\star$	0.3504	**0.5363**$\diamondsuit \star \bullet$	0.3684	0.2864
Metabolism	-0.784	-0.6912$\star$	60.00	**64.76**$\diamondsuit \star \bullet$	56.51	59.57
Carcinogenesis	-0.6712	-0.6618	59.38	**61.30**$\diamondsuit \bullet$	55.0	55.70

From the table we can see that BFORTE, by revising the structure, can im-

prove the probabilities of examples, since conditional log-likelihood has significant increased in two of the three cases. The final value of the score function is better than the first-order systems in all cases, showing that it is possible to obtain more accurate systems when uncertainty is taken into account. Moreover, in two cases the revision in structure improved the initial score. Although learning parameters has improved the initial score of Carcinogenesis compared to first-order systems, the final score is not significantly better than the initial score.

8.4.2 Speed up in the revision process due to the Bottom clause

Now, we would like to verify whether introducing the Bottom Clause as space of literals can decrease the runtime without harming the score. To focus on this issue, we run BFORTE using the Bottom Clause procedure and BFORTE using FOIL algorithm to generate literals. Results of runtime and score are exhibited in Table 8.2. As expected, UW-CSE performs significantly faster when the Bottom Clause is the search space of literals. Surprisingly, the score is also better in the Bottom Clause case, although the difference is not significant. By analyzing the results of each fold, we found out that one fold has a higher score for the FOIL case, as it adds to a clause one literal that has not appeared in the Bottom Clause of the chosen example in BFORTE case. A similar situation also happens in another fold, but this time the literal added to the clause in one of the iterations makes the theory performs worse in the validation set. As a result, BFORTE has a better score for this fold.

Unfortunately, we were not able to collect FOIL results in the other two datasets, as the system runs out of memory after some time running. The reason, besides the much larger search space generated for FOIL approach, is that differently from UW-CSE, Metabolism and Carcinogenesis have several predicates with constants defined in mode declarations. The top-down approach of FOIL cannot generate literals with constants, and then a variable is put in the place of a possible constant. The problem that arises is the large number of instantiations of the same literal, yielding a huge amount of different proof paths for the same literal. Consider, for example, the following mode definition of a predicate in Carcinogenesis

domain.

$modeb(*, atm(+drug, -atomid, \#element, \#integer, -charge))$.

Types *element* and *integer* are defined as constants. Bottom Clause construction procedure is able to find a constant to the place of such mode, since it is created from a particular instance. Those constants are going to limit the ground facts that can be unified with this literal. However, FOIL puts variables in third and fourth terms. As a consequence, every possible $atm/5$ ground literal in the dataset is able to unify with the variabilized new literal, producing in certain cases a huge amount of different proofs that Matlab cannot handle. Note that, this is an additional complexity of BLPs compared to the first-order case, as in this last it is not necessary to collect all possible proofs explaining an example.

Table 8.2: Comparison of runtime and accuracy of BFORTE with Bottom Clause and BFORTE using FOIL to generate literals.

	BFORTE		BFORTE without BC	
Dataset	Learning Time	Score	Learning Time	Score
UW-CSE	8061.07⋆	0.5363	10884.16	0.5159
Metabolism	7184.15	64.76	N/A	N/A
Carcinogenesis	8812.16	61.30	N/A	N/A

8.4.3 Selection of Revision Points

In this experiment, we focus on the question of whether the BFORTE approach to select revision points can decrease learning time. Due to the limitations of PFORTE and BLP, specially concerning the search space of literals (previous section), we compare BFORTE to BFORTE simulating those algorithms with regard to the selection of revision points. Notice that this is necessary, specially because those systems cannot run in two cases without bounding the search space to the Bottom Clause, as we can see in Table 8.2. The compared settings are as follows.

- BFORTE: this is the system implemented upon all the contributions presented in this chapter: the Bottom clause is used to bound the search space of lit-

erals, revision points are selected using Bayes Ball and inference procedure is optimized.

- PFORTE-like: this case considers Bottom clause to bound the search space and optimized inference. However, selection of the revision points follows PFORTE system, where all clauses appearing in the network of a misclassified instance are marked as revision points.

- BLP-like: this case considers Bottom clause to bound the search space and optimized inference. However, selection of the revision points follows BLP structure learning algorithm, where all Bayesian clauses in the BLP are refined.

We call the last two cases as PFORTE-like and BLP-like because original PFORTE and BLP learning algorithm *have none of the improvements* we designed in this chapter.

Table 8.3 shows the results of runtime, accuracy and number of revised clauses for each setting. The symbol $\star$ indicates the cases where there is significant difference between BFORTE and PFORTE-like and symbol $\bullet$ indicates a significant difference between BFORTE and BLP-like. Runtime of BFORTE is significantly better than both PFORTE-like and BLP-like. Note that PFORTE performs worse than BLP. This is due to the size of the network: as the network of UW-CSE is quite large and PFORTE selects all clauses taking part in a network of a misclassified instance, it always marks the whole theory as revision points. Thus, it has the same search space as BLPs. However, as BLPs modifies all Bayesian clauses and PFORTE performs inference to find out the misclassified instances, it expends more time to finish the revision process than BLP-like. On the other hand, the score of BFORTE is worse than PFORTE and BLP, although not significantly. The reason for that difference is that in one fold different theories are kept by the validation set during the learning.

Unfortunately, we cannot obtain results in reasonable time using either PFORTE or BLP settings for Metabolism dataset ($< 48h$). This is mainly due to the large search space that is explored, since, during the learning time that we could trace, they mark 18 clauses on average to be modified, which is the same number of clauses in the initial theory. The same situation of UW-CSE happens here com-

Table 8.3: Comparison of BFORTE runtime and score with PFORTE and BLP algorithms for selecting points to me modified.

System/ Dataset	BFORTE Learning Time	BFORTE Score	BFORTE #Revised clauses	PFORTE-like Learning Time	PFORTE-like Score	PFORTE-like #Revised clauses	BLP-like Learning Time	BLP-like Score	BLP-like #Revised clauses
UW-CSE	8061.07⋆ •	0.5363	4.7	10517.49	0.5669	7.6	10734.23	0.5669	7.6
Metab	7184.15	-64.76	6.3	N/A	N/A	18	N/A	N/A	18
Carcino	8812.16 ⋆	61.30	3.8	8812.16	61.30	3.8	26922.49	61.76	8.0

paring PFORTE and BLP: as this dataset yields a single large network, PFORTE marks every clause to be revised. BFORTE considers only 6.3 clauses on average to be revised, which is due to the smart selection of revision points conducted by Bayes Ball algorithm.

A different situation happens with Carcinogenesis. As the instances are not related in this dataset, there is a Bayesian network for each instance, yielding smaller individual networks than in previous cases. In this way, BFORTE and PFORTE-like select the same clauses as revision points. This is also because the clauses producing ascendant nodes to the top-level instances are derived from clauses in the fixed background knowledge, and therefore cannot be modified. Additionally, the ground facts produced by them are considered as non-observed. Hence, even though they were modifiable, they would non-observed parent nodes whose parents are observed and therefore Bayes Ball would also consider them as revision points. Finally, we see that runtime of BLP-like is much worse than revision systems, since in this case it alone marks all clauses from the theory as revision points.

8.4.4 Inference time

Finally, we show the reduction in runtime of the inference due to the use of Bayes Ball and the grouping of clauses and networks. Note that all cases previously discussed takes into account inference with Bayes Ball when learning or revising theories. Instead of running the whole revision process, we opted for running only the procedure that computes the discriminative score for each training set, considering the initial theories. In this way, we are able to see the *improvement in runtime inference alone*,

without taking into account the particularities of the revision process. Additionally, running original *inference* of PFORTE and BLP makes the revision/learning process extremely slow.

Tables 8.4, 8.5 and 8.6 shows the runtime for computing accuracy in UW-CSE, Metabolism and Carcinogenesis, respectively, of five settings: BFORTE, with all improvements presented in section 8.3, BFORTE without collecting requisite nodes with Bayes Ball, BFORTE without detecting similar networks, BFORTE without overlapping same rules and inference performed in original PFORTE/BLP i.e, inference is performed without any of the algorithms presented in section 8.3. Let first focus on UW-CSE dataset. Observe that the runtime is reduced by a factor of 170 from the full BFORTE setting. The largest reduction is due to Bayes Ball, although grouping similar nodes also contributes. On the other hand, detecting similar networks does not help in the reduction of the runtime. In two folds Matlab runs out of memory for PFORTE/BLP. Metabolism dataset has similar results of UW-CSE, since they both are highly relational datasets, but in this case we were able to collect the values for all training sets.

Table 8.4: Inference runtime in seconds for UW-CSE dataset.

Setting/ Training set	BFORTE	BFORTE, without Bayes Ball	BFORTE without similar nets	BFORTE without grouping rules	PFORTE/ BLP
1	18.44	1247.72	20.28	74.62	N/A
2	4.16	89.36	4.40	15.34	443.62
3	4.56	92.80	4.74	23.92	693.98
4	22.86	1880.38	25.30	97.94	N/A
5	3.44	66.76	3.52	15.00	443.72

Table 8.5: Inference runtime in seconds for Metabolism dataset.

Setting/ Training set	BFORTE	BFORTE, without Bayes Ball	BFORTE without similar nets	BFORTE Without Grouping rules	PFORTE/ BLP
1	10.98	677.62	676.90	23.30	1505.58
2	9.16	541.82	542.44	19.90	1103.22
3	13.02	827.56	826	29.92	1990.48
4	8.24	487.14	485.78	15.68	958.98
5	11.44	697.44	696.70	21.30	1330.40

The inference runtime of Carcinogenesis it is not different from PFORTE and

BLP. In fact, in most of the cases, the runtime is slightly worse, due to the cost of executing Bayes Ball. However, the fourth training set has a particular behavior when running BFORTE without grouping rules. In this case, one of the Bayesian clauses yields a large amount of ground clauses to be part of the final network. Without grouping similar rules, the final Bayesian network is too large for Matlab to handle.

Table 8.6: Inference runtime in seconds for Carcinogenesis dataset.

Setting/ Training set	BFORTE	BFORTE, without Bayes Ball	BFORTE without similar nets	BFORTE Without Grouping rules	PFORTE/ BLP
1	5.24	4.36	4.12	6.66	5.08
2	5.34	4.44	4.30	7.50	5.78
3	4.64	3.92	3.70	6.04	4.68
4	9.70	8.96	8.64	N/A	N/A
5	4.48	3.76	3.46	9.48	6.96

8.5 Conclusions

We addressed in this chapter the bottlenecks of Bayesian Logic Programs revision. We showed through experiments that it is possible to obtain a feasible revision system that provides more accurate models than first-order logic systems, when the domain is uncertain.

First we focused on the reduction of new literals search space. The baseline system, PFORTE, generates literals following a FOIL's top down approach: all possible literals from the language that had at least one common variable with the current clause were considered to be added to a clause. We were able to reduce the search space of literals by defining it as the Bottom Clause of a misclassified instance. We show in the experiments that the search space is reduced without harming the score achieved by the system. As a single misclassified instance may not carry sufficient information to produce good literals, in the future we would like to investigate the use of more several instances to create the Bottom Clause. Additionally, we could also take into account the probabilistic information to create/remove literals from the Bottom clause, for example by measuring the information that literals could pass to the clause (Cheng et al., 2002a; Oliphant and Shavlik, 2008; Pitangui and

Zaverucha, 2011). The work in (Mihalkova and Mooney, 2007) employs Bottom-up learning, by considering the network of examples to reduce the search space of possible refinements. We intend to investigate how such an approach could also be followed when refining Bayesian Logic Programs. It does not seem that there is a direct way of doing that, since we construct the Bayesian networks from the ground clauses, instantiated by the examples, whereas Markov Logical Networks defines the graph from the constants in the domain.

Next, we addressed the search space of clauses to be refined. BLP learning algorithm starts from an initial set of Bayesian clauses and proposes modifications to each one. PFORTE considers all the clauses used to build the Bayesian network of an example where there was a misclassified instance. Both systems generate a large search space that can even become intractable. We showed that it is possible to reduce this search space by marking as revision points only the clauses that influence the probability distribution computed for the misclassified instance. We used the Bayes Ball algorithm to identify those clauses. Experiments suggested that the search space is indeed reduced by proposing refinements only to the clauses relevant to the misclassified instance. In the future, we would like to investigate in more details if it is possible to reduce the number of revision operators according to the revision point, as we started to analyze in this chapter. Additionally, search for revision points guided by examples may not identify all possible places bringing new influence to a misclassified instance. In the future, we intend to investigate whether it is possible to have a good balance on efficiency and score when choosing revision points in promising places, but still outside the set pointed out by the instance.

Finally, we addressed the large inference search space due to the large Bayesian networks produced by PLL examples. First, we developed a procedure to only consider the requisite nodes identified by Bayes Ball algorithm, when performing inference. Second, we reduced the size of the network by identifying ground clauses whose only difference are the terms replacing variables. We argue that it is not necessary to represent them as separate nodes in the network, what would make the inference procedure to repeat several probabilities computations. Third, we identified that after selecting the requisite nodes, there are "subnetworks" with

exactly the same structure and evidences. We avoid to compute the probabilities for all those networks, by taking into account only one representative of the group. Experiments showed that the inference runtime is in fact greatly reduced by the algorithms we proposed. Note that the we only took a simple step in the direction of recent work on lifted inference (Poole, 2003; Salvo Braz et al., 2005; Singla and Domingos, 2008; Milch et al., 2008; Kok and Domingos, 2009; Nath and Domingos, 2010; Meert et al., 2010). There is much more to be done and investigate to fulfill the advantages of lifted inference. For example, the ground clauses that are grouped together could represent more than one Bayesian clause, if the information they have are the same.

Through preliminary experiments, we noticed that the revision greatly depends on the initial theories. Aleph system may not be appropriate to learn initial theories for giving rise to BLPs. Indeed, the authors of BLP argued that Claudien system (De Raedt and Dehaspe, 1997), which learns from interpretations, is more adequate to produce most general clauses. In the future we intend to create several different theories, from different first-order systems and compare the revised BLPs, starting from them.

In the experiments reported here, the threshold for marking an instance as misclassified was set at 0.5, which is usually considered in binary domains. That value may not be the best for evaluating the models. Thus, we intend to create curves showing the behavior of the system with different thresholds. Also, an evaluation function that is independent from the threshold should be applied, such as the ones considering the area under the curve. Last, it is essential to compare BLPs to others SRL languages, such as Markov Logic Networks.

We conclude by observing that as stochastic local search has been showed to be quite effective in the first-order revision case, we intend to implement in BFORTE the same SLS techniques considered in YAVFORTE system.

Chapter 9

General Conclusions and Future work

This chapter summarizes the achievements of the thesis, presents work in progress that we have been investigating and points directions for future research.

9.1 Summary

The goal of this thesis was to contribute with effective theory revision systems. Arguably, theory revision has been set aside so far due to the cost of searching for revisions in large search space. With this thesis we have shown that it is possible to have a theory revision system running in feasible time and achieving better accuracies than learning from scratch approaches.

In chapter 3 we contributed with a revision system named YAVFORTE, that had been built over FORTE (Richards and Mooney, 1995) system. YAVFORTE offers to the user the choice of selecting revision operators, so that the number of possible modifications to be proposed to a theory is decreased. We empirically showed that there are several cases where a smaller set of revision operators can achieve same accuracies than if all operators were in use, in a reduced runtime. Also, revision operators are employed from the simplest to more complex, so that if a simpler operator is already able to reach the maximum potential of a revision point, it is not necessary to appeal to complex operators. Another important contribution implemented as part of YAVFORTE is the use of the Bottom Clause and mode declarations to reduce the search space of literals to be scored. This greatly reduced the runtime of the revision process, compared to the original FORTE system.

In chapter 4 we designed a challenging application of theory revision involving

the game of Chess. We developed a framework for acquiring the rules of variants of chess, starting from the rules of traditional chess. It was necessary to include further abduction strategies and also handle negated literals in theories under revision. We showed that theory revision is able to yield theories describing variants of Chess, while a system unable to modify the initial theory fails. This work also showed that theory revision is able to handle transfer learning tasks where the predicates of the original and final models are the same.

Although YAVFORTE is able to revise theories faster than an inductive system learns theories in several cases, this is not always the case. If the dataset has a large number of examples and/or background knowledge, and moreover, the initial theory has a large number of clauses, the revision process would struggle. Thus, in chapter 6 we devised a series of stochastic local search algorithms, implemented on each key search of the revision process, so that good solutions may be found instead of optimal ones but in reasonable time. The system built upon YAVFORTE and including SLS was named ASSERTE. By randomizing revision points, revision operators and literals to be added/removed to/from a clause we achieved faster learning time and equivalent accuracies compared to YAVFORTE. In the best case, a SLS algorithm executed $25X$ faster than the baseline revision process. Moreover, better accuracies and competitive runtime are also achieved, compared to a standard inductive learning system.

In chapter 8 we contributed with a feasible Bayesian Logic Programs revision system. We addressed the bottlenecks of our previous revision system PFORTE that made it impractical to handle real world datasets. First, following our algorithm FORTE_MBC (Duboc et al., 2009), we defined the Bottom Clause to be the search space of new literals, either when a hill climbing approach or the relational pathfinding algorithm is employed. Next, we addressed revision points selection, by employing Bayes Ball algorithm (Shachter, 1998), so that the nodes influencing a wrongly predicted instance are identified and the clauses that produced them are revised. Revision operators intend to either change the present influence or bring new influence through those points, so that instances become correctly classified. Last, we developed a procedure to speed up the time expended by the revision pro-

cess to perform inference. First, we required that only requisite nodes are taken into account when computing the probability of a node. The set of requisite nodes is also identified by Bayes Ball algorithm. Second, we overlap nodes present in ground clauses of an instance, that had been instantiated from the same clause and also have the same evidence. Finally, we identify sets of requisite nodes that are identical, in order to avoid performing repeated inference for different instances. We show that all these optimizations are able to reduce the runtime of the revision process, by comparing each one of them to the process followed by PFORTE and BLP learning algorithm.

9.2 Future work: First-Order Logic Theory Revision

First-order revision runtime may be further reduced by using techniques developed in ILP. For example, strategies transforming queries, devised in (Santos Costa et al., 2003b) may greatly reduce inference time, by making clauses more efficient to evaluate. Also, there is need for efficient scoring of the hypothesis, specially in the presence of a large set of examples. This can be achieved by more efficient subsumption tests (Kuzelka and Zelezný, 2008a; Kuzelka and Zelezný, 2008b; Santos and Muggleton, 2011) and by incremental learning techniques (Lopes and Zaverucha, 2009).

Concerning SLS algorithms, their runtime may greatly vary depending on the various random choices taken along the revision process. To avoid being stuck for a long time in a search space when looking for a specific modification to be done in the theory, strategies that *restarts* the search from a different points raises the opportunity of abandoning a large and unproductive search and starts over from a perhaps more promising seed. The strategy followed in (Železný et al., 2006) can be applied to the revision case, so that after a number of hypothesis are scored without success, the search procedure chooses another random point to restart. Additionally, stochastic search may also be used to estimate the coverage of clauses (Sebag and Rouveirol, 1997).

There still need for real and challenging applications that fits well with theory revision. We tackle this issue by developing the Chess revision framework. Applica-

tions from grammar learning, natural language processing and biology, containing significant coded information, seem also to be good examples of problems for theory revision to handle.

9.3 Future work and work in progress: Probabilistic Logic Revision

9.3.1 Revision of Bayesian Logic Programs

Inference performed in Bayesian networks may dominate the runtime of the revision process, since this is the most executed task in the system. We showed in this thesis that inference runtime can be greatly reduced by avoiding repeated computations and by reducing the size of the network. It is necessary to further explore sophisticated techniques developed in the field of lifted inference in probabilistic logical models (Poole, 2003; Kok and Domingos, 2009; Kersting et al., 2010), so that inference runtime is reduced even more and hence the revision process. Moreover, similar to the first-order case, stochastic local search may be show very useful to reduce the runtime of the probabilistic revision process.

In this thesis we have addressed the revision from a *discriminative* point of view: instances in examples are classified and in case there are wrongly predicted instances, Bayesian clauses relevant to the misclassification are revised. The question that arises is if the revision can be performed from a *generative* approach. In this case, the need of revision would be indicated by, for example, a low generative score. Examples would not be separated in query variables and evidences, but instead they could be all in the same level, as it is done in *learning from interpretations* setting.

9.3.2 Revision of Stochastic Logic Programs

Bayesian Logic Programs encode probability distributions over possible worlds by associating probability distributions to possible values of atoms. Stochastic Logic Programs (Muggleton, 2002), on the other hand, follows a distributional semantics based on domain frequency (Halpern, 1989) by associating probability distributions to ground facts through stochastic proof trees. A SLP is composed of a set of stochastic clauses in the form $p : c$, where c is a definite range restricted clause

and p is a probability label. Summing out probability labels of all clauses with the same predicate in the head must produce the value 1. SLPs combine definite logic programs with Probabilistic Context Free Grammar, by generalizing this last one.

We have designed a system named SCULPTOR for revising Stochastic Logic Programs. Examples in SCULPTOR have a probability label as in (Chen et al., 2008) and this label is used to compute root mean squared error (RMSE). Examples with high RMSE are chosen to indicate the clauses that should be revised. Revision operators include revision of the probability labels, deletion and addition of rules. New clauses are created by unfolding existing clauses (Sato, 1992), guided by a top theory (Muggleton et al., 2008).

A first prototype of SCULPTOR has already been implemented, but it needs to be tested and experimentally evaluated. We want to apply SCULPTOR on the challenge application of revising strategies of games.

In closing, I would like to briefly cite the main achievements of this thesis, which are: (1) to make the the Theory Revision from Examples as efficient as ILP; (2) to successfully apply Theory Revision from Examples in a Chess application, where standard ILP fails; and (3) to improve a Bayesian Logic Programs revision system we had previously developed so that it could be applied to real world problems. I believe that the revision of (probabilistic) first-order logical theories has the potential to handle complex real world applications and reach better results than simpler machine learning thecniques. With this thesis, we showed that this can be achieved in feasible time.

Bibliography

Adé, H., Malfait, B., and De Raedt, L. (1994). RUTH: an ILP Theory Revision System. In *8th International Symposium on Methodologies for Intelligent Systems (ISMIS-94)*, volume 869 of *LNCS*, pages 336–345. Springer.

Allen, T. and Greiner, R. (2000). Model Selection Criteria for Learning Belief Nets: an Empirical Comparison. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 1047–1054.

Alphonse, É. and Rouveirol, C. (2000). Lazy Propositionalisation for Relational Learning. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00)*, pages 256–260.

Angluin, D., Frazier, M., and Pitt, L. (1992). Learning Conjunctions of Horn Clauses. *Machine Learning*, 9:147–164.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice.* Oxford University, New York.

Baffes, P. (1994). *Automatic Student Modeling and Bug Library Construction using Theory Refinement.* PhD thesis, University of Texas, Austin, TX.

Baffes, P. and Mooney, R. (1993). Symbolic Revision of Theories with M-of-N Rules. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1135–1140, Chambery, France.

Baião, F., Mattoso, M., Shavlik, J., and Zaverucha, G. (2003). Applying Theory Revision to the Design of Distributed Databases. In *Proceedings of the 13th*

International Conference on Inductive Logic Programming (ILP-03), volume 2835 of *LNAI*, pages 57–74. Springer.

Bain, M. (2004). Predicate Invention and the Revision of First-Order Concept Lattice. In *2nd International Conference on Formal Concept Analysis (ICFCA 2004)*, volume 2961 of *LNAI*, pages 329–336. Springer.

Bain, M. and Muggleton, S. (1994). Learning Optimal Chess strategies. *Machine Intelligence*, 13:291–309.

Battle, A., Segal, E., and Koller, D. (2004). Probabilistic Discovery of Overlapping Cellular Processes and their Regulation. In *RECOMB*, pages 167–176.

Binder, J., Koller, D., Russell, S. J., and Kanazawa, K. (1997). Adaptive Probabilistic Networks with Hidden Variables. *Machine Learning*, 29(2-3):213–244.

Blockeel, H. and De Raedt, L. (1998). Top-Down Induction of First-Order Logical Decision Trees. *Artificial Intelligence*, 101(1-2):285–297.

Booth, J. G. and Hobert, J. P. (1999). Maximizing Generalized Linear Mixed Model Likelihoods with an Automated Monte Carlo EM Algorithm. *Journal of the Royal Statistical Society, Series B*, 61:265–285.

Bratko, I. (1999). Refining Complete Hypotheses in ILP. In *Proceedings of the 9th Inductive Logic Programming (ILP-99)*, volume 1634 of *LNAI*, pages 44–55. Springer.

Bratko, I. and King, R. D. (1994). Applications of Inductive Logic Programming. *SIGART Bulletin*, 5(1):43–49.

Bunescu, R. C. and Mooney, R. J. (2004). Collective Information Extraction with Relational Markov Networks. In *ACL*, pages 438–445.

Buntine, W. (1991). Theory Refinement on Bayesian Networks. In *Proceedings of the 17th Annual Conference on Uncertainty in Artificial Intelligence (UAI-91)*, pages 52–60, San Mateo, CA.

Burg, D. B. and Just, T. (1987). *IUS Chess Federation Official Rules of Chess*. McKay, New York, USA.

Caffo, B. S., Jank, W. S., and Jones, G. L. (2005). Ascent-Based Monte Carlo EM. *Journal of the Royal Statistical Society, Series B*, 67:235 – 252.

Cai, D. M., Gokhale, M., and Theiler, J. (2007). Comparison of Feature Selection and Classification Algorithms in Identifying Malicious Executables. *Computational Statistics and Data Analysis*, 51(6).

Campos, L. M. (2006). A Scoring Function for Learning Bayesian Networks based on Mutual Information and Conditional Independence Tests. *Journal of Machine Learning Research*, 7:2149–2187.

Caruana, R. (1997). Multitask Learning. *Machine Learning*, 28(1):41–75.

Celeux, G., Chauveau, D., and Diebolt, J. On Stochastic Versions of the EM Algorithm. Technical Report RR-2514.

Celeux, G. and Diebolt, J. (1985). The SEM algorithm: a Probabilistic Teacher Algorithm Derived from the EM Algorithm for the Mixture Problem. *Comput. Statist. Quater.*, 2:73–82.

Cerny, V. (1985). A Thermodynamical Approach to the Traveling Salesman Problem. *Journal of Optimization Theory and Applications*, 45(1):41–51.

Chan, D. (1988). Constructive Negation Based on the Completed Database. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 111–125. The Mit Press.

Chen, J., Muggleton, S., and Santos, J. C. A. (2008). Learning Probabilistic Logic Models from Probabilistic Examples. *Machine Learning*, 73(1):55–85.

Cheng, J., Greiner, R., Kelly, J., Bell, D. A., and Liu, W. (2002a). Learning Bayesian Networks from Data: An Information-Theory Based Approach. *Artificial Intelligence*, 137(1-2):43–90.

Cheng, J., Hatzis, C., Hyashi, H., Krogel, M.-A., Morishita, S., Page, D., and Sese, L. (2002b). KDD Cup 2001 report. *SIGKDD Explorations*, 3(2):47–64.

Chickering, D. M. (2002). Optimal Structure Identification With Greedy Search. *Journal of Machine Learning Research*, 3:507–554.

Chickering, M. and Heckerman, D. (1997). Efficient Approximations for the Marginal Likelihood of Bayesian Networks with Hidden Variables. *Machine Learning*, 29:181–212.

Chisholm, M. and Tadepalli, P. (2002). Learning Decision Rules by Randomized Iterative Local Search. In *Proceedings of the 19th International Conference on Machine Learning (ICML-02)*, pages 75–82.

Clark, K. (1978). Negation as Failure Rule. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 293–322. Plenum Press.

Cohen, W. (1992). Compiling Prior Knowledge into an Explicit Bias. In *Proceedings of the Nineteenth International Conference on Machine Learning (ICML-92)*, pages 102–110, Aberdeen, Scotland.

Cohen, W. (1995). Fast Effective Rule Induction. In *Proceedings of 12th International Conference on Machine Learning (ICML-95)*, pages 115–123. Morgan Kaufmann.

Collet, P. and Rennard, J.-P. (2007). Stochastic Optimization Algorithms. *CoRR*, abs/0704.3780.

Cooper, G. F. (1990). The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence*, 42(2-3):393–405.

Cowell, R. G., Dawid, A. P., Lauritzen, S. L., and Spiegelhalter, D. J. (1999). *Probabilistic Networks and Expert Systems*. Springer, New York.

Cussens, J. (2001). Parameter Estimation in Stochastic Logic Programs. *Machine Learning*, 44(3):245–271.

Dagum, P. and Luby, M. (1993). Approximating Probabilistic Inference in Bayesian Belief Networks is NP-Hard. *Artificial Intelligence*, 60(1):141–153.

Darwiche, A. (2010). Bayesian networks. *Communications of the ACM*, 53(12):80–90.

David Heckerman, C. M. and Koller, D. (2004). Probabilistic Models for Relational Data. Technical Report MSR-TR-2004-30, Microsoft Research.

Davis, J., Burnside, E. S., de Castro Dutra, I., Page, D., Ramakrishnan, R., Santos Costa, V., and Shavlik, J. W. (2005a). View Learning for Statistical Relational Learning: With an Application to Mammography. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 677–683.

Davis, J., Burnside, E. S., de Castro Dutra, I., Page, D., and Santos Costa, V. (2005b). An Integrated Approach to Learning Bayesian Networks of Rules. In *Proceedings of the 16th European Conference on Machine Learning (ECML-05)*, volume 3720 of *LNAI*, pages 84–95. Springer.

Davis, J. and Goadrich, M. (2006). The Relationship between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML-06)*, pages 233–240.

Davis, J., Ong, I. M., Struyf, J., Burnside, E. S., Page, D., and Santos Costa, V. (2007). Change of Representation for Statistical Relational Learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2719–2726.

De Raedt, L. (1996). *Advances in Inductive Logic Programming.* IOS Press.

De Raedt, L. (1997). Logical Settings for Concept-Learning. *Artificial Intelligence*, 95(1):187–201.

De Raedt, L. (2008). *Logical and Relational Learning.* Springer, Berlin, German.

De Raedt, L. and Bruynooghe, M. (1993). A Theory of Clausal Discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1058–1063.

De Raedt, L. and Dehaspe, L. (1997). Clausal Discovery. *Machine Learning*, 2(3):99–146.

De Raedt, L. and Džeroski, S. (1994). First-Order jk-Clausal Theories are PAC-Learnable. *Artificial Intelligence*, 70(1-2):375–392.

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors (2008a). *Probabilistic Inductive Logic Programming – Theory and Applications*, volume 4911 of *LNAI*. Springer.

De Raedt, L. and Kersting, K. (2004). Probabilistic Inductive Logic Programming. In *Proceedings of the 15th International Conference on Algorithmic Learning Theory (ALT-2004), Invited paper in S. Ben-David, J. Case and A. Maruoka, editors*, pages 19–36.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., and Toivonen, H. (2008b). Compressing Probabilistic Prolog Programs. *Machine Learning*, 70(2-3):151–168.

De Raedt, L., Kersting, K., and Torge, S. (2005). Towards Learning Stochastic Logic Programs from Proof-Banks. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 752–757.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007). ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2462–2467.

DeGroot, M. H. (1989). *Probability and Statistics*. Addison-Wesley.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38.

Dietterich, T. (1998). Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation*, 10:1895–1924.

Dietterich, T., Domingos, P., Getoor, L., Muggleton, S., and Tadepalli, P. (2008). Structured Machine learning: The Next Ten Years. *Machine Learning*, 73:3–23.

Díez, F. J. and Galán, S. F. (2002). Efficient Computation for the Noisy Max. *International Journal of Intelligent Systems*, 18:165–177.

Dimopoulos, Y. and Kakas, A. (1995). Learning Non-Monotonic Logic Programs: Learning Exceptions. In *Proceedings of the 8th European Conference on Machine Learning (ECML-95)*, volume 912 of *LNAI*, pages 122–138. Springer.

Domingos, P. and Lowd, D. (2009). *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Domingos, P. and Pazzani, M. J. (1997). On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29(2-3):103–130.

Drabent, W. (1995). What is Failure? An Approach to Constructive Negation. *Acta Inf.*, 32(1):27–29.

Drabent, W. (1996). Completeness of SLDNF-Resolution for Nonfloundering Queries. *Journal of Logic Programming*, 27(2):89–106.

Duboc, A. L. (2008). Tornando a Revisão de Teorias de Primeira-ordem mais Eficiente Através da Utilização da Cláusula Mais Específica. Master's thesis, COPPE - UFRJ, Rio de Janeiro, RJ.

Duboc, A. L., Paes, A., and Zaverucha, G. (2008). Using the Bottom Clause and Modes Declarations on FOL Theory Revision from Examples. In *Proceedings of the 18th International Conference on ILP (ILP-08)*, volume 5194 of *LNAI*, pages 91–106. Springer.

Duboc, A. L., Paes, A., and Zaverucha, G. (2009). Using the Bottom Clause and Modes Declarations on FOL Theory Revision from Examples. *Machine Learning*, 76(1):73–107.

Dzeroski, S. and Bratko, I. (1992). Handling Noise in Inductive Logic Programming. In *Proceedings of the 2nd International Workshop on Inductive Logic Programming*.

Dzeroski, S. and Lavrac, N., editors (2001). *Relational Data Mining*. Springer-Verlag.

Elidan, G. and Friedman, N. (2005). Learning Hidden Variable Networks: The Information Bottleneck Approach. *Journal of Machine Learning Research*, 6:81–127.

Elidan, G., Lotner, N., Friedman, N., and Koller, D. (2000). Discovering Hidden Variables: A Structure-Based Approach. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS-00)*, pages 479–485. MIT Press.

Esposito, F., Semeraro, G., Fanizzi, N., and Ferilli, S. (2000). Multistrategy Theory Revision: Induction and Abduction in INTHELEX. 38:133–156.

Fang, H., Tong, W., Shi, L. M., Blair, R., Perkins, R., Branham, W., Hass, B. S., Xie, Q., Dial, S. L., Moland, C. L., and Sheehan, D. M. (2001). Structure-Activity Relationships for a Large Diverse Set of Natural, Synthetic, and Environmental Estrogens. *Chemical Research in Toxicology*, 3(14):280–294.

Feller, W. (1970). *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, third edition.

Fensel, D., Zickwolff, M., and Wiese, M. (1995). Are Substitutions the Better Examples? In *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP-95)*, pages 120–127.

Feo, T. and Resende, M. (1995). Greedy Randomised Adaptive Search Procedures. *Journal of Global Optimization*, (6):109–133.

FIDE. FIDE Laws of Chess.

Fierens, D., Blockeel, H., Bruynooghe, M., and Ramon, J. (2005). Logical Bayesian Networks and their Relation to Other Probabilistic Logical Models. In *15th International Conference on Inductive Logic Programming*, pages 121–135.

Flach, P. (1994). *Simply Logical - Intelligent Reasoning by Example*. John Wiley & Sons.

Flach, P. and Kakas, A. (2000a). *Abduction and Induction: Essays on their Relation and Integration*. Kluwer Academic Publishers.

Flach, P. and Kakas, A. (2000b). Abductive and Inductive Reasoning: Background and Issues. In Flach, P. A. and Kakas, A. C., editors, *Abductive and Inductive Reasoning*. Kluwer Academic Publishers.

Fogel, L. and Zaverucha, G. (1998). Normal Programs and Multiple Predicate Learning. In *Proceedings of the 8th Int. Conf. on ILP*, volume 1446 of *LNAI*, pages 175–184. Springer.

Frank, E. and Witten, I. H. (1998). Generating Accurate Rule Sets Without Global Optimization. In *Proceedings of 15th International Conference on Machine Learning (ICML-98)*, pages 144–151. Morgan Kaufmann.

Frazier, M. and Pitt, L. (1993). Learning From Entailment: An Application to Propositional Horn Sentences. In *Proceedings of the Tenth International Conference on Machine Learning (ICML-93)*, pages 120–127.

Friedman, N. (1998). The Bayesian Structural EM Algorithm. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 129–138. Morgan Kaufmann.

Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian Network Classifiers. *Machine Learning*, 29:131–163.

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning Probabilistic Relational Models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1300–1309.

Fürnkranz, J. (1996). Machine Learning in Computer Chess: The Next Generation. *Int. Computer Chess Association Journal*, 19(3):147–161.

Fürnkranz, J. (2007). Recent Advances in Machine Learning and Game Playing. *OGAI-Journal*, 26(2):147–161.

Garcez, A. and Zaverucha, G. (1999). The Connectionist Inductive Learning and Logic Programming System. *Applied Intelligence*, 11:59–77.

Geiger, D., Verma, T., and Pearl, J. (1989). d-Separation: From Theorems to Algorithms. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence (UAI-89)*, pages 139–148. North-Holland.

Geiger, D., Verma, T., and Pearl, J. (1990). Identifying independence in Bayesian networks. *Networks*, 20:507–534.

Geman, S. and Geman, D. (1984). Stochastic Relaxation, Gibbs Distributions and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741.

Getoor, L. (2007). An introduction to Statistical Relational Learning. In *Tutorial Notes of the 18th European Conference on Machine Learning and the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*.

Getoor, L., Friedman, N., Koller, D., and Taskar, B. (2001). Learning Probabilistic Models of Relational Structure. In *Proceedings of the 8th International Conference on Machine Learning (ICML-01)*, pages 170–177.

Getoor, L. and Grant, J. (2006). PRL: A probabilistic Relational Language. *Machine Learning*, 62(1-2):7–31.

Getoor, L., Rhee, J. T., Koller, D., and Small, P. (2004). Understanding Tuberculosis Epidemiology Using Structured Statistical Models. *Artificial Intelligence in Medicine*, 30(3):233–256.

Getoor, L. and Taskar, B., editors (2007). *An Introduction to Statistical Relational Learning*. MIT Press.

Gilks, W., Richardson, S., and Spiegelhalter, D. (1995). *Markov Chain Monte Carlo in Practice: Interdisciplinary Statistics*. Chapman & Hall/CRC.

Glover, F. (1989). Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206.

Glover, F. (1990). Tabu Search - Part II. *INFORMS Journal on Computing*, 2(1):4–32.

Goadrich, M., Oliphant, L., and Shavlik, J. W. (2004). Learning Ensembles of First-Order Clauses for Recall-Precision Curves: A Case Study in Biomedical Information Extraction. In *Proceedings of the 14th International Conference on ILP (ILP-04)*, volume 3194 of *LNAI*, pages 98–115. Springer.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Gomes, C. P., Selman, B., Crato, N., and Kautz, H. (2000). Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1-2):67–100.

Gomes, C. P., Selman, B., McAloon, K., and Tretkoff, C. (1998). Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. In *AIPS*, pages 208–213.

Goodacre, J. (1996). Master thesis, Inductive Learning of Chess Rules Using Progol. Programming Research Group, Oxford University.

Greiner, R. and Zhou, W. (2002). Structural Extension to Logistic Regression: Discriminative Parameter Learning of Belief Net Classifiers. In *Proceedings*

of the 18th Annual National Conference on Artificial Intelligence (AAAI-02), pages 167–173.

Grossman, D. and Domingos, P. (2004). Learning Bayesian Network Classifiers by Maximizing Conditional Likelihood. In *Proceedings of the 21st International Conference on Machine Learning (ICML-04)*, pages 361–368.

Gu, J. (1992). Efficient Local Search for Very Large-scale Satisfiability Problems. *SIGART Bulletin*, 3(1):8–12.

Gyftodimos, E. and Flach, P. A. (2004). Hierarchical Bayesian Networks: An Approach to Classification and Learning for Structured Data. In *Methods and Applications of Artificial Intelligence, 3rd Helenic Conference on AI (SETN 2004)*, volume 3025 of *LNAI*, pages 291–300. Springer.

Haddaway, P. (1999). An Overview of Some Recent Developments on Bayesian Problem Solving Techniques. *AI Magazine - Special issue on Uncertainty in AI*, 20(2):11–29.

Hall, M. A. (2000). Correlation-Based Feature Selection for Discrete and Numeric Class Machine Learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 359–366.

Hall, M. A. and Smith, L. A. (1999). Feature Selection for Machine Learning: Comparing a Correlation-Based Filter Approach to the Wrapper. In *Proceedings of the 12th International Florida Artificial Intelligence Research Society Conference*, pages 235–239.

Halpern, J. Y. (1989). An Analysis of First-Order Logics of Probability. *Artificial Intelligence*, 46:311–350.

Heckerman, D. (1996). A Tutorial on Learning with Bayesian Networks. Technical Report MSR-TR-95-06, Microsoft Research, http://research.microsoft.com/ heckerman/.

Heckerman, D. and Breese, J. (1994). Causal Independence for Probability Assessment and Inference using Bayesian Networks. Technical Report MSR-TR-94-08, Microsoft Research, http://research.microsoft.com/ heckerman/.

Hilden, J. (1984). Statistical Diagnosis Based on Conditional Independence does not Require it. *Compu. Biol. Med.*, 14(4):429–435.

Hirst, J. D., King, R. D., and Sternberg, M. J. E. (1994a). Quantitative Structure-Activity Relationships by Neural Networks and Inductive Logic Programming. I. The Inhibition of Dihydrofolate Reductase by Pyrimidines. *Journal of Computer-Aided Molecular Design*, 8(4):405–420.

Hirst, J. D., King, R. D., and Sternberg, M. J. E. (1994b). Quantitative Structure-Activity Relationships by Neural Networks and Inductive Logic Programming. I. The Inhibition of Dihydrofolate Reductase by Triazines. *Journal of Computer-Aided Molecular Design*, 8(4):421–432.

Hooper, D. and Whyld, K. (1992). *The Oxford Companion to Chess*. Oxford University Press.

Hoos, H. H. and Stützle, T. (2005). *Stochastic Local Search: Foundations and Applications*. Elsevier, California, USA, 1 edition.

Hutter, F., Hoos, H. H., and Stützle, T. (2005). Efficient Stochastic Local Search for MPE Solving. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 169–174.

Huynh, T. N. and Mooney, R. J. (2008). Discriminative Structure and Parameter Learning for Markov Logic Networks. In *Proceedings of the 25th International Conference on Machine Learning (ICML-08)*, volume 307 of *ACM International Conference Proceeding Series*, pages 416–423. ACM.

Inoue, K. (2001). Induction, Abduction, and Consequence-Finding. In *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *LNCS*, pages 65–79. Springer.

Jaeger, M. (1997). Relational Bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 266–273. Morgan Kaufmann.

Jaimovich, A., Elidan, G., Margalit, H., and Friedman, N. (2005). Towards an Integrated Protein-Protein Interaction Network. In *RECOMB*, pages 14–30.

Jank, W. (2006). The EM Algorithm, Its Randomized Implementation and Global Optimization: Some Challenges and Opportunities for Operations Research. In Francis B. Alt, M. C. F. and Golden, B. L., editors, *Perspectives in Operations Research*, pages 367–392. Springer.

Jensen, D. and Neville, J. (2002). Linkage and Autocorrelation Cause Feature Selection Bias in Relational Learning. In *Proceedings of the Nineteenth International Conference on Machine Learning (ICML-02)*, pages 259–266. Morgan Kaufmann.

Jensen, D., Neville, J., and BrianGallagher (2004). Why Collective Inference Improves Relational Classification. In *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 593–598. ACM.

Jensen, F. V. (1996). *An introduction to Bayesian networks*. Springer, New York.

Jensen, F. V. (2001). *Bayesian Networks and Decision Graphs*. Springer-Verlag.

John, G. H., Kohavi, R., and Pfleger, K. (1994). Irrelevant Features and the Subset Selection Problem. In *Proceedings of the 11th International Conference on Machine Learning (ICML-94)*, pages 121–129.

Joshi, S., Ramakrishnan, G., and Srinivasan, A. (2008). Feature Construction Using Theory-Guided Sampling and Randomised Search. In *Proceedings of the 18th International Conference on ILP*, pages 140–157.

Zuželka, O. and Železný, F. (2007). A Restart Strategy for Fast Subsumption Check and Coverage Estimation. In *Proceedings of the 6th Workshop on Multi-Relational Data Mining (MRDM 2007)*.

Kadupitige, S. R., Julia, K. C. L., Sellmeier, Sivieng, J., Catchpoole, D. R., Bain, M., and Gaeta, B. A. (2009). MINER: Exploratory Analysis of Gene Interaction Networks by Machine Learning from Expression Data. *BMC Genomics*, 10(Suppl 3):S17.

Kautz, H., Selman, B., and Jiang, Y. (1996). A General Stochastic Approach to Solving Problems with Hard and Soft Constraints. In Du, D., Gu, J., and Pardalos, P. M., editors, *The Satisfability Problem: Theory and Applications (DIMACS Workshop March 11-13, 1996.)*.

Kearns, M. J. and Mansour, Y. (1998). Exact Inference of Hidden Structure from Sample Data in noisy-OR Networks. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 304–310. Morgan Kaufmann.

Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. Cambridge, Massachusetts.

Kersting, K. (2006). *An Inductive Logic Programming Approach to Statistical Relational Learning*, volume 148 of *Frontiers in Artificial Intelligence and its Applications series Dissertations*. IOS Press.

Kersting, K. and De Raedt, L. (2001a). Adaptive Bayesian Logic Programs. In *Proceedings of the 11th Conference on Inductive Logic Programming (ILP-01)*, volume 2157 of *LNAI*, pages 104–117. Springer.

Kersting, K. and De Raedt, L. (2001b). Bayesian Logic Programs. *CoRR*, cs.AI/0111058.

Kersting, K. and De Raedt, L. (2001c). Bayesian Logic Programs. Technical Report 151, University of Freiburg, Institute for Computer Science, http://www.informatik.uni-freiburg.de/ kersting/publications.html.

Kersting, K. and De Raedt, L. (2001d). Towards Combining Inductive Logic Programming with Bayesian Networks. In *Proceedings of the 11th Conference on*

Inductive Logic Programming (ILP-01), volume 2157 of *LNAI*, pages 118–131. Springer.

Kersting, K. and De Raedt, L. (2002). Basic Principles of Learning Bayesian Logic Programs. Technical Report 174, University of Freiburg, Institute for Computer Science, http://www.informatik.uni-freiburg.de/ kersting/publications.html.

Kersting, K. and De Raedt, L. (2007). Bayesian logic programming: Theory and tool. In Getoor, L. and Taskar, B., editors, *An Introduction to Statistical Relational Learning*, pages 291–322.

Kersting, K., De Raedt, L., and Raiko, T. (2006). Logical Hidden Markov Models. *Journal of Artificial Intelligence Research (JAIR)*, 25:425–456.

Kersting, K., Massaoudi, Y. E., Hadiji, F., and Ahmadi, B. (2010). Informed Lifting for Message-Passing. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*. AAAI Press.

Kersting, K., Raiko, T., Kramer, S., and De Raedt, L. (2003). Towards Discovering Structural Signatures of Protein Folds Based on Logical Hidden Markov Models. In *Pacific Symposium on Biocomputing*, pages 192–203.

Kimmig, A. (2010). *A Probabilistic Prolog and its Applications.* PhD thesis, Katholieke Universiteit Leuven, Belgium.

Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and Raedt, L. D. (2008). On the Efficient Execution of ProbLog Programs. In *24th International Conference on Logic Programming (ICLP-08)*, volume 5366 of *LNCS*, pages 175–189. Springer.

King, R. D., Muggleton, S., and Sternberg, M. (1992). Drug Design by Machine Learning: The Use of Inductive Logic Programming to Model the Structure-Activity Relationships of Trimethoprim Analogues Binding to Dihydrofolate Reductase. *Proceedings of the National Academy of Sciences*, 89(23):11322–11326.

King, R. D., Sternberg, M. J. E., and Srinivasan, A. (1995a). Relating Chemical Activity to Structure: An Examination of ILP Successes. *New Generation Computing*, 13(3-4):411–433.

King, R. D., Sternberg, M. J. E., and Srinivasan, A. (1995b). Relating Chemical Activity to Structure: An Examination of ILP successes. *New Generation Computing*, 13(3-4):411–433.

King, R. D., Whelan, K., Jones, F., Reiser, P., Bryant, C., Muggleton, S. H., Kell, D., and Oliver, S. (2004). Functional Genomic Hypothesis Generation and Experimentation by a Robot Scientist. *Nature*, 427:247–252.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, 220:671–680.

Kohavi, R. (1995). A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence(IJCAI-95)*, pages 1137–1145.

Kok, S. and Domingos, P. (2005). Learning the Structure of Markov Logic Networks. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-05)*, volume 119 of *ACM International Conference Proceeding Series*, pages 441–448. ACM.

Kok, S. and Domingos, P. (2007). Statistical Predicate Invention. In *Proceedings of the 24th International Conference on Machine Learning (ICML-07)*, pages 433–440. ACM Press.

Kok, S. and Domingos, P. (2009). Learning Markov Logic Network Structure via Hypergraph Lifting. In *Proceedings of the 26th Annual International Conference on Machine Learning, (ICML-09)*, volume 382 of *ACM International Conference Proceeding Series*, page 64. ACM.

Koller, D. (1999). Probabilistic Relational Models. In *Proceedings of the 9th International Conference on Inductive Logic Programming (ILP-99)*, pages 3–13.

Koller, D. and Friedman, N. (2009). *Probabilistc Graphical Models: Principles and Techniques*. MIT Press.

Koller, D. and Pfeffer, A. (1997). Learning Probabilities for Noisy First-Order Rules. In *Proceedings of the 15th International Joint Conference on Artficial Intelligence (IJCAI-97)*, pages 1316–1323.

Koller, D. and Pfeffer, A. (1998). Probabilistic Frame-Based Systems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 580–587.

Koller, D. and Sahami, M. (1996). Toward Optimal Feature Selection. In *Proceedings of the 13th International Conference on Machine Learning (ICML-96)*, pages 284–292.

Koppel, M., Feldman, R., and Segre, A. M. (1994). Bias-Driven Revision of Logical Domain Theories. *Journal of Artificial Intelligence Research*, 1:159–208.

Kovacic, M., Lavrac, N., Grobelnik, M., Zupanic, D., and Mladenic, D. (1992). Stochastic Search in Inductive Logic Programming. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-92)*, pages 444–445.

Kramer, S. (1995). Predicate Invention: A Comprehensive View. Technical Report FAI-TR-95-32, Austrian Research Institute for Artificial Intelligence.

Krogel, M.-A., Rawles, S., Železný, F., Flach, P. A., Lavrac, N., and Wrobel, S. (2003). Comparative Evaluation of Approaches to Propositionalization. In *Proceedings of the 13th ILP*, volume 2835 of *LNAI*, pages 197–214. Springer.

Kuzelka, O. and Zelezný, F. (2008a). A Restarted Strategy for Efficient Subsumption Testing. *Fundam. Inform.*, 89(1):95–109.

Kuzelka, O. and Zelezný, F. (2008b). Fast Estimation of First-Order Clause Coverage Through Randomization and Maximum Likelihood. In *Proceedings of the 25th International Conference on Machine Learning (ICML-08)*, volume 307 of *ACM International Conference Proceeding Series*, pages 504–511. ACM.

Kuzelka, O. and Zelezný, F. (2011). Seeing the World through Homomorphism: An Experimental Study on Reducibility of Examples. In *Revised Papers of the 20th International Conference on Inductive Logic Programming (ILP-10)*, volume 6489 of *LNAI*, pages 138–145. Springer.

Laarhoven, P. J. V. and Arts, E. H. L. (1987). *Simulated Annealing: Theory and Applications*. Reidel Publishers, Amsterdam.

Lachiche, N. and Flach, P. A. (2002). 1BC2: A True First-Order Bayesian Classifier. In *Proceedings of the 12th International Conference on Inductive Logic Programming (ILP-02)*, volume 2583 of *LNAI*, pages 133–148. Springer.

Lam, W. and Bacchus, F. (1994). Learning Bayesian Belief Networks: an Approach Based on the MDL Principle. *Computational Intelligence*, 10(4):269–293.

Landwehr, N., Kersting, K., and De Raedt, L. (2007). nFOIL: Integrating Naïve Bayes and FOIL. 8:481–507.

Landwehr, N., Passerini, A., Raedt, L. D., and Frasconi, P. (2006). kFOIL: Learning Simple Relational Kernels. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*.

Langley, P. (1995). *Elements of Machine Learning*. Morgan Kaufman.

Langley, P., Iba, W., and Thompson, K. (1992). An Analysis of Bayesian Classifiers. In *Proceedings of the 8th Annual National Conference on Artificial Intelligence (AAAI-92)*, pages 223–228.

Lauritzen, S. L. (1995). The EM algorithm for graphical Association Models with Missing Data. *Computational Statistics and Data Analysis*, 19:191–201.

Lavrac, N. and Dzeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York.

Lavrac, N. and Dzeroski, S. (2001). *Relational Data Mining*. Springer, New York.

Levy, D. N. L. and Newborn, M. (1990). *How Computers Play Chess*.

Li, Z. and D'Ambrosio, B. (1994). Efficient Inference in Bayes Networks as a Combinatorial Optimization Problem. *International Journal of Approximate Reasoning*, 11(1):55–81.

Lloyd, J. (1987). *Foundations of Logic Programming*. Springer, second edition.

Lopes, C. and Zaverucha, G. (2009). HTILDE: Scaling up Relational Decision Trees for Very Large Databases. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1475–1479. ACM.

Lourenćo, H. R., O.Martin, and Stützle, T. (2002). Iterated Local Search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers.

Marriott, K., Søndergaard, H., and Dart, P. (1990). A Characterization of Non-Floundering Logic Programs. In *Proceedings of the 1990 North American conference on Logic programming*, pages 661–680. MIT Press.

McCulloch, C. E. (1994). Maximum Likelihood Variance Components Estimation for Binary Data. *Journal of the American Statistical Association*, 89:330–335.

McLachlan, G. J. and Krishnan, T. (1997). *The EM algorithm and Extensions*. Wiley Interscience, New York, 1 edition.

Meert, W., Taghipour, N., and Blockeel, H. (2010). First-Order Bayes-Ball. In *Proceedings of the 22nd ECML/PKDD 2*, LNAI 6322, pages 369–384. Springer.

Michalski, R. S. and Larson, J. (1977). Inductive Inference of VL decision rules. In *Workshop in pattern-Directed Inference Systems, SIGART Newsletter*, volume 63, pages 38–44. ACM.

Mihalkova, L., Huynh, T., and Mooney, R. J. (2007). Mapping and Revising Markov Logic Networks for Transfer Learning. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 608–614.

Mihalkova, L. and Mooney, R. J. (2007). Bottom-up learning of Markov logic network structure. In *Proceedings of the 24h International Conference on Machine*

Learning (ICML-07), volume 227 of *ACM International Conference Proceeding Series*, pages 625–632. ACM.

Milch, B., Marthi, B., Russell, S. J., Sontag, D., Ong, D. L., and Kolobov, A. (2005). BLOG: Probabilistic Models with Unknown Objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1352–1359.

Milch, B., Zettlemoyer, L. S., Kersting, K., Haimes, M., and Kaelbling, L. P. (2008). Lifted Probabilistic Inference with Counting Formulas. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, (AAAI-08)*, pages 1062–1068. AAAI Press.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, USA.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, New York.

Mooney, R. (1997). Integrating Abdcution and Induction in Machine Learning. In *Workshop on Abduction and Induction in AI, IJCAI-97*, pages 37–42.

Mooney, R. (2000). Integrating Abduction and Induction in Machine Learning. In Flach, P. A. and Kakas, A. C., editors, *Abduction and Induction*, pages 181–191. Kluwer Academic Publishers.

Moyle, S. (2003). Using Theory Completion to Learn a Robot Navigation Control Program. In *Proceedings of 12th International Conference on ILP*, volume 2583 of *LNAI*, pages 182–197. Springer.

Muggleton, S. (1987). Duce, An Oracle-based Approach to Constructive Induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 287–292.

Muggleton, S. (1991). Inductive Logic Programming. *New Generation Computing*, 13(4):245–286.

Muggleton, S. (1992). *Inductive Logic Programming*. Academic Press, New York.

Muggleton, S. (1994). Predicate Invention and Utilisation. *Journal of Experimental and Theoretical Artficial Intelligence*, 6(1):127–130.

Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing*, 13(3&4):245–286.

Muggleton, S. (1996). Stochastic Logic Programs. In Raedt, L. D., editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press.

Muggleton, S. (1999). Scientific Knowledge Discovery Using Inductive Logic Programming. *Communications of the ACM*, 42(11):42–46.

Muggleton, S. (2000). Learning Stochastic Logic Programs. *Electron. Trans. Artif. Intell.*, 4(B):141–153.

Muggleton, S. (2002). Learning Structure and Parameters of Stochastic Logic Programs. In *Proceedings of the 12th International Conference on Inductive Logic Programming (ILP-02)*, volume 2583 of *LNAI*, pages 198–206. Springer.

Muggleton, S. (2005). Machine Learning for Systems Biology. In *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP-05)*, volume 3625 of *Lecture Notes in Computer Science*, pages 416–423. Springer.

Muggleton, S. and Bryant, C. H. (2000). Theory Completion Using Inverse Entailment. In *Proceedings of the 10th International Conference on ILP*, volume 1866 of *LNAI*, pages 130–146. Springer.

Muggleton, S. and Buntine, W. L. (1988). Machine Invention of First Order Predicates by Inverting Resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352.

Muggleton, S. and De Raedt, L. (1994). Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19(20):629–679.

Muggleton, S. and Feng, C. (1990). Efficient Induction of Logic Programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory (COLT-90)*, pages 368–381.

Muggleton, S., Paes, A., Santos Costa, V., and Zaverucha, G. (2009a). Chess Revision: Acquiring the Rules of Chess Variants through FOL Theory Revision from Examples. In *Proceedings of the IJCAI-09 Workshop on General Game Playing General Intelligence in Game-Playing Agents (GIGA'09)*, pages 61–71.

Muggleton, S., Paes, A., Santos Costa, V., and Zaverucha, G. (2009b). Chess Revision: Acquiring the Rules of Chess Variants through FOL Theory Revision from Examples. In *Proceedings of the 19th International Conference on ILP (ILP-09)*, volume 5989 of *LNAI*, pages 123–130. Springer.

Muggleton, S., Paes, A., Santos Costa, V., and Zaverucha, G. (2009c). Chess Revision: Acquiring the Rules of Chess Variants through FOL Theory Revision from Examples. In *Workshop on Transfer Leaning - NIPS 2009*.

Muggleton, S. and Pahlavi, N. (2007). Stochastic logic program: A tutorial. In Getoor, L. and Taskar, B., editors, *An Introduction to Statistical Relational Learning*, pages 323–338.

Muggleton, S., Santos, J. C. A., and Tamaddoni-Nezhad, A. (2008). TopLog: ILP Using a Logic Program Declarative Bias. In *Proceedings of the 24th International Conference on Logic Programming (ICLP-08)*, pages 687–692.

Muggleton, S., Santos, J. C. A., and Tamaddoni-Nezhad, A. (2010). ProGolem: A System Based on Relative Minimal Generalisation. In *Proceedings of the 1th International Conference on Inductive Logic Programming (ILP-09)*, pages 131–148.

Muggleton, S. and Tamaddoni-Nezhad, A. (2008). QG/GA: a stochastic search for Progol. *Machine Learning*, 70(2-3):121–133.

Muggleton, S. H., King, R. D., and Sternberg, M. J. E. (1992). Protein Secondary Structure Prediction Using Logic-Based Machine Learning. *Protein Engineering*, 5(7):647–657.

Murphy (1997). Bayes Net Toolbox for Matlab. U.C. Berkeley.

Murphy, K. (2001). The Bayes Net Toolbox for Matlab. *Computing Science and Statistics*, 33.

Murphy, K. P., Weiss, Y., and Jordan, M. I. (1999). Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 467–475. Morgan Kaufmann.

Nadeau, C. and Bengio, Y. (2003). Inference for the Generalization Error. *Machine Learning*, 53(3):239–281.

Natarajan, S., Tadepalli, P., Dietterich, T. G., and Fern, A. (2008). Learning First-Order Probabilistic Models with Combining Rules. *Annals of Mathematical Artificial Intelligence*, 54(1-3):223–256.

Nath, A. and Domingos, P. (2010). Efficient Lifting for Online Probabilistic Infe-rence. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, (AAAI-10)*. AAAI Press.

Neumann, J., Schnörr, C., and Steidl, G. (2005). Combined SVM-Based Feature Selection and Classification. *Machine Learning*, 61(1-3):129–150.

Neville, J. and Jensen, D. (2004). Dependency Networks for Relational Data. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM-24)*, pages 170–177.

Neville, J., Simsek, Ö., Jensen, D., Komoroske, J., Palmer, K., and Goldberg, H. G. (2005). Using Relational Knowledge Discovery to Prevent Securities Fraud. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-05)*, pages 449–458. ACM.

Ngo, L. and Haddawy, P. (1997). Answering Queries from Context-Sensitive Probabilistic Knowledge Bases. *Theoretical Computer Science*, 171(1-2):147–177.

Nielsen, S. (2000). The Stochastic EM Algorithm: Estimation and Asymptotic Results. *Journal of the Bernoulli Society for Mathematical Statistics and Probability*, 6(3):457–489.

Nienhuys-Cheng, Shan-Hwei, and De Wolf, R. (1997). *Foundations of Inductive Logic Programming*. Springer-Verlag.

Nienhuys-Cheng, S.-H. and De Wolf, R. (1997). *Foundations of Inductive Logic Programming*. Springer, Berlin.

Nilsson, U. and Maluszynkski, J. (2000). *Logic Programming and PROLOG*. John Wiley and Sons, 2nd edition.

Oliphant, L. and Shavlik, J. W. (2008). Using Bayesian Networks to Direct Stochastic Search in Inductive Logic Programming. In *Revised Selected Papers of the 17th International Conference on Inductive Logic Programming (ILP-07)*, volume 4894 of *LNAI*, pages 191–199. Springer.

Ong, I. M., Dutra, I. C., Page, D., and Santos Costa, V. (2005). Mode Directed Path Finding. In *Proceedings of the 16th ECML*, volume 3720, pages 673–681.

Ourston, D. and Mooney, R. J. (1994). Theory Refinement Combining Analytical and Empirical Methods. *Artificial Intelligence*, (66):311–344.

Paes, A., Revoredo, K., Zaverucha, G., and Santos Costa, V. (2005a). Further Experimental Results of Probabilistic First-order Revision of Theories from Examples. In *Workshop on Multi-Relational Data Mining*. SIGKDD.

Paes, A., Revoredo, K., Zaverucha, G., and Santos Costa, V. (2005b). Probabilistic First-Order Theory Revision from Examples. In *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP-05)*, volume 3625 of *LNAI*, pages 295–311. Springer.

Paes, A., Revoredo, K., Zaverucha, G., and Santos Costa, V. (2006a). PFORTE: Revising Probabilistic FOL Theories. In *Proceedings of the 18th Brazilian AI Symposium (SBIA-06)*, volume 4140 of *LNAI*, pages 441–450. Springer.

Paes, A., Zaverucha, G., and Santos Costa, V. (2007a). Revisando Teorias Lógicas de Primeira-ordem a partir de Exemplos usando Busca Local Estocástica. In *Anais do VI Encontro Nacional de Inteligência Artificial ENIA-07. Prêmio de*

melhor artigo do congresso. XXVII Congresso Brasileiro de Computação SBC 2007. SBC.

Paes, A., Zaverucha, G., and Santos Costa, V. (2007b). Revising First-order Logic Theories from Examples through Stochastic Local Search. In *Proceedings of the 17th International Conference on ILP (ILP-07)*, volume 4894 of *LNAI*, pages 200–210. Springer.

Paes, A., Železný, F., Zaverucha, G., Page, D., and Srinivasan, A. (2006b). ILP through Propositionalization and Stochastic k-term DNF Learning. In *Proceedings of the Revised Papers of 16th International Conference on ILP (ILP-06)*, volume 4455 of *LNAI*, pages 379–393. Springer.

Paes, A. M. (2005). PFORTE - Revisão de Teorias Probabilísticas de Primeira-ordem através de Exemplos. Master's thesis, COPPE - UFRJ, Rio de Janeiro, RJ.

Page, D. and Srinivasan, A. (2003). ILP: A Short Look Back and a Longer Look Forward. *Journal of Machine Learning Research*, 4:415–430.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Francisco, CA.

Pearl, J. (1991). *Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, second edition.

Pearl, J. (2009). *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition.

Pellet, J.-P. and Elisseeff, A. (2008). Using Markov Blankets for Causal Structure Learning. *Journal of Machine Learning Research*, 9:1295–1342.

Peot, M. A. and Shachter, R. D. (1991). Fusion and Propagation with Multiple Observations in Belief Networks. *Artificial Intelligence*, 48(3):299–318.

Pina, A. C. and Zaverucha, G. (2004). An Algorithmic Presentation of Accuracy Comparison of Classification Learning Algorithms. Technical report, Universidade Federal do Rio de Janeiro, COPPE/PESC, Rio de Janeiro, Brasil.

Pitangui, C. G. and Zaverucha, G. (2011). Inductive Logic Programming through Estimation of Distribution Algorithm. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC-2011)*, pages 54–61. IEEE.

Plotkin, G. D. (1971). *Automatic Methods of Inductive Inference*. PhD thesis.

Poole, D. (1993). Probabilistic Horn Abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129.

Poole, D. (1997). The Independent Choice Logic for Modelling Multiple Agents Under Uncertainty. *Artificial Intelligence*, 94(1-2):7–56.

Poole, D. (2003). First-order Probabilistic Inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 985–991. Morgan Kaufmann.

Poole, D. (2008). The Independent Choice Logic and Beyond. In De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors, *Probabilistic Inductive Logic Programming – Theory and Applications*, volume 4911 of *LNAI*, pages 222–243. Springer.

Pritchard, D. B. (2007). *The Classified Encyclopedia of Chess Variants*. John Beasley.

Provost, F. J., Fawcett, T., and Kohavi, R. (1998). The Case Against Accuracy Estimation for Comparing Induction Algorithms. In *Proceedings of the 15th International Conference on Machine Learning (ICML-98)*, pages 445–453. Morgan Kaufmann.

Quinlan, J. (1990). Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266.

Raghavan, S., Kovashka, A., and Mooney, R. (2010). Authorship Attribution Using Probabilistic Context-Free Grammars. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 38–42. The Association for Computer Linguistics.

Ramachandran, S. and Mooney, R. (1998). Theory Refinement of Bayesian Networks with Hidden Variables. In *Proceedings of the 15th International Conference on Machine Learning (ICML-98)*, pages 454–462.

Ray, O. (2009). Nonmonotonic Abductive Inductive Learning. *Journal of Applied Logic*, 7(3):329–340.

Ray, O., Broda, K., and Russo, A. (2004). A Hybrid Abductive Inductive Proof Procedure. *Logic Journal of the IGPL*, 12(5):371–397.

Revoredo, K. (2009). *PFORTE_PI: Revisão de Teorias Probabilísticas Relacionais com Invenção de Predicados*. PhD thesis.

Revoredo, K., Paes, A., Zaverucha, G., and Santos Costa, V. (2006). Combining Predicate Invention and Revision of Probabilistic FOL Theories. In *Short paper proceedings of 16th International Conference on Inductive Logic Programming (ILP-07)*, pages 176–178.

Revoredo, K., Paes, A., Zaverucha, G., and Santos Costa, V. (2007). Combinando Invenção de predicados e revisão de teorias de primeira-ordem probabilísticas. In *Anais do VI Encontro Nacional de Inteligência Artificial. XXVII Congresso Brasileiro de Computação SBC 2007*. SBC.

Revoredo, K., Paes, A., Zaverucha, G., and Santos Costa, V. (2009). Revisando Redes Bayesianas através da Introdução de Variáveis não-observadas. In *Anais do VII Encontro Nacional de Inteligência Artificial ENIA-09. XXIX Congresso Brasileiro de Computação SBC 2009*. SBC.

Revoredo, K. and Zaverucha, G. (2002). Revision of First-Order Bayesian Classifiers. In *Proceedings of the 12th International Conference on ILP (ILP-02)*, pages 223–237.

Richards, B. L. and Mooney, R. J. (1992). Learning Relations by Pathfinding. In *Proceedings of the 10th Annual National Conference on Artificial Intelligence (AAAI-92)*, pages 50–55.

Richards, B. L. and Mooney, R. J. (1995). Automated Refinement of First-Order Horn-Clause Domain Theories. *Machine Learning*, 19(2):95–131.

Richardson, M. and Domingos, P. (2006). Markov Logic Networks. *Machine Learning*, 62(1-2):107–136.

Ross, S. M. (1988). *A First Course in Probability*. Macmillan, third edition.

Rückert, U. and Kramer, S. (2003). Stochastic Local Search in k-Term DNF Learning. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 648–655.

Rückert, U. and Kramer, S. (2004). Towards Tight Bounds for Rule Learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML-04)*, volume 69. ACM.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A modern approach*. Englewook Cliffs, NJ: Prentice-Hall, 3rd edition.

Sadikov, A. and Bratko, I. (2006). Learning Long-Term Chess strategies from databases. *Machine Learning*, 63(3):329–340.

Salvo Braz, R., Amir, E., and Roth, D. (2005). Lifted First-Order Probabilistic Inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1319–1325. Professional Book Center.

Salvo Braz, R., Amir, E., and Roth, D. (2006). MPE and Partial Inversion in Lifted Probabilistic Variable Elimination. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press.

Sang, T., Beame, P., and Kautz, H. A. (2007). A Dynamic Approach for MPE and Weighted MAX-SAT. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 173–179.

Santos, J. and Muggleton, S. (2011). When Does It Pay Off to Use Sophisticated Entailment Engines in ILP? In *Revised Papers of the 20th International Conference on Inductive Logic Programming (ILP-10)*, volume 6489 of *LNAI*, pages 214–221. Springer.

Santos Costa, V. (2008). The Life of a Logic Programming System. In *Proceedings of the 24th International Conference on Logic Programming (ICLP-08)*, volume 5366 of *LNCS*, pages 1–6. Springer.

Santos Costa, V. and Paes, A. (2009). On the Relationship between PRISM and CLP(BN). In *Proceedings of the International Workshop on Statistical Relational Learning (SRL-09)*.

Santos Costa, V., Page, D., Qazi, M., and Cussens, J. (2003a). CLPBN: Constraint Logic Programming for Probabilistic Knowledge. In *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 517–524.

Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., and Laer, W. V. (2003b). Query Transformations for Improving the Efficiency of ILP Systems. *Journal of Machine Learning Research*, 4:465–491.

Sato, T. (1992). Equivalence-Preserving First Order Unfold/fold Transformation Systems. *Theoretical Computer Science*, 105:57–84.

Sato, T. and Kameya, Y. (1997). PRISM: A Language for Symbolic-Statistical Modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1330–1339.

Sato, T. and Kameya, Y. (2001). Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research JAIR*, 15:391–454.

Sato, T., Kameya, Y., and Zhou, N.-F. (2005). Generative Modeling with Failure in

PRISM. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 847–852.

Schwarz, G. (1978). Estimating the Dimension of a Model. *Annals of Statistics*, 6:461–464.

Sebag, M. and Rouveirol, C. (1997). Tractable Induction and Classification in First Order Logic Via Stochastic Matching. In *IJCAI (2)*, pages 888–893.

Sebag, M. and Rouveirol, C. (2000). Any-time Relational Reasoning: Resource-bounded Induction and Deduction Through Stochastic Matching. *Machine Learning*, 38(1-2):41–62.

Selman, B., Kautz, H., and Cohen, B. (1994). Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 337–343. AAAI Press/The MIT Press.

Selman, B., Kautz, H., and McAllester, D. (1997). Ten Challenges in Propositional Reasoning and Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 50–54. Morgan Kaufmann Publishers.

Selman, B. and Kautz, H. A. (1993). Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295.

Selman, B., Kautz, H. A., and Cohen, B. (1996). Local Search Strategies for Satisfiability Testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:521–532.

Selman, B., Levesque, H., and Mitchell, D. (1992). A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th Annual National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446.

Serrurier, M. and Prade, H. (2008). Improving Inductive Logic Programming by Using Simulated Annealing. *Information Sciences*, 178(6):1423–1441.

Shachter, R. D. (1998). Bayes-Ball: The Rational Pastime for Determining Irrelevance and Requisite Information in Belief Networks and Influence Diagrams. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 480–487. Morgan Kaufmann.

Shapiro, E. Y. (1981). The Model Inference System. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI-81)*, page 1064. William Kaufmann.

Shapiro, E. Y. (1983). *Algorithm Program Debugging*. MIT Press.

Singla, P. and Domingos, P. (2005). Discriminative Training of Markov Logic Networks. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-05)*, pages 868–873.

Singla, P. and Domingos, P. (2006). Memory-Efficient Inference in Relational Domains. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*.

Singla, P. and Domingos, P. (2008). Lifted First-Order Belief Propagation. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, (AAAI-08)*, pages 1094–1099. AAAI Press.

Specia, L., Srinivasan, A., Joshi, S., Ramakrishnan, G., and das Graças Volpe Nunes, M. (2009). An Investigation into Feature Construction to Assist Word Sense Disambiguation. *Machine Learning*, 76(1):109–136.

Spellman, P., Sherlock, G., Zhang, M., Iyer, V., Anders, K., Eisen, M., Brown, P., D.Botstein, and B.Futcher (1998). Comprehensive Identification of Cell Cycle-Regulated Genes of the Yeast Saccharomyces Cerevisiae by Microarray Hybridization. *Molecular Biology of the cell*, 9(12):3273–3297.

Spirtes, P., Glymour, C., and Scheines, R. (2001). *Causation, Prediction, and Search*. MIT Press, 2nd edition.

Srinivasan, A. (1999). A Study of Two Sampling Methods for Analysing Large Datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123.

Srinivasan, A. (2000). A Study of Two Probabilistic Methods for Searching Large Spaces with ILP. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, Oxford.

Srinivasan, A. (2001a). Extracting Context-Sensitive Models in Inductive Logic Programming. *Machine Learning*, 3(44):301–324.

Srinivasan, A. (2001b). *The Aleph Manual*. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html`.

Srinivasan, A. and King, R. D. (2008). Incremental Identification of Qualitative Models of Biological Systems using Inductive Logic Programming. *Journal of Machine Learning Research*, 9:1475–1533.

Srinivasan, A., King, R. D., Muggleton, S., and Sternberg, M. J. E. (1997). Carcinogenesis Predictions Using ILP. In *Proceedings of the 7th International Conference on Inductive Logic Programming (ILP-97)*, volume 1297 of *Lecture Notes in Computer Science*, pages 273–287. Springer.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., and King, R. D. (1996). Theories for Mutagenicity: A Study in First-Order and Feature-Based Induction. *Artificial Intelligence*, 85(1-2):277–299.

Srinivasan, A., Page, D., Camacho, R., and King, R. D. (2006). Quantitative Pharmacophore Models with Inductive Logic Programming. *Machine Learning*, 64(1-3):65–90.

Stahl, I. (1993). Predicate Invention in ILP - an Overview. In *Proceedings of the 4th European Conference on Machine Learning (ECML 94)*, volume 667 of *LNCS*, pages 313–322. Springer.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques*. The MIT Press.

Stone, M. (1977). An asymptotic Equivalence of Choice of Model by Cross-Validation and Akaike's Criterion. *Journal of the Royal Statistical Society series B*, 39:44–47.

Tamaddoni-Nezhad, A., Chaleil, R., Kakas, A. C., and Muggleton, S. (2006). Application of Abductive ILP to Learning Metabolic Network Inhibition from Temporal Data. *Machine Learning*, 64(1-3):209–230.

Tamaddoni-Nezhad, A. and Muggleton, S. (2000). Searching the Subsumption Lattice by a Genetic Algorithm. In *Proceedings of the 10th International Conference on ILP (ILP-00)*, pages 243–252.

Tang, L. R., Mooney, R. J., and Melville, P. (2003). Scaling Up ILP to Large Examples: Results on Link Discovery for Counter-Terrorism. In *Proceedings of the KDD-2003 Workshop on Multi-Relational Data Mining (MRDM-2003)*, pages 107–121.

Taskar, B., Abbeel, P., and Koller, D. (2002). Discriminative Probabilistic Models for Relational Data. In *Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence (UAI-02)*, pages 485–492.

Thrun, S. (1995). Is Learning the n-th Thing any Easier than Learning the First? In *Advances in Neural Information Processing Systems 8, NIPS*, pages 640–646. MIT Press.

Tian, J. (2000). A Branch-and-Bound Algorithm for MDL Learning Bayesian Networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, UAI '00, pages 580–588. Morgan Kaufmann Publishers Inc.

Tian, J. and Pearl, J. (2002). On the Testable Implications of Causal Models with Hidden Variables. In *Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence (UAI-02)*, pages 519–527. Morgan Kaufmann.

Titov, I. and Henderson, J. (2007). Incremental Bayesian Networks for Structure Prediction. In *Proceedings of the 24th International Conference on Machine Learning (ICML-07)*, volume 227 of *ACM International Conference Proceeding Series*, pages 887–894. ACM.

Towell, G. and Shavlik, J. (1994). Knowledge-Based Artificial Neural Networks. *Artificial Intelligence*, 70(1–2):119–165.

Towell, G. G. and Shavlik, J. W. (1993). Extracting Refined Rules from Knowledge-Based Neural Networks. *Machine Learning*, 13:71–101.

Trefethen, N. (1998). *Maxims about Numerical Mathematics, Computers, Science, and Life*. SIAM News.

Tsamardinos, I., Brown, L. E., and Aliferis, C. F. (2006). The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm. *Machine Learning*, 65(1):31–78.

Valiant, L. G. (1984). A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142.

Van Rijsbergen, C. J. (1979). *Information Retrieval*. Butterworths.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. (2004). Logic Programs with Annotated Disjunctions. In *Proceedings of the 20th International Conference on Logic Programming (ICLP-04)*, pages 431–445.

Verma, T. and Pearl, J. (1990). Equivalence and synthesis of causal models. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence*, pages 255–270. Elsevier.

Wang, J. and Domingos, P. (2008). Hybrid Markov Logic Networks. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 1106–1111. AAAI Press.

Wei, G. C. G. and Tanner, M. A. (1990). A Monte Carlo Implementation of the EM Algorithm and the Poor Man's Data Augmentation Algorithms. *Journal of the American Statistical Association*, 85:699–704.

Weiss, Y. (2000). Correctness of Local Probability Propagation in Graphical Models with Loops. *Neural Computation*, 12(1):1–41.

Weston, J., Mukherjee, S., Chapelle, O., Pontil, M., Poggio, T., and Vapnik, V. (2000). Feature Selection for SVMs. In *Papers from Neural Information Processing Systems (NIPS-00)*, pages 668–674.

Wogulis, J. (1994). *An Approach to Repairing and Evaluationg First-Order Theories Containing Multiple Concepts and Negation.* PhD thesis, University of California, Irvine, CA.

Wogulis, J. and Pazzani, M. (1993). A methodology for Evaluationg Theory Revision Systems: Results with Audrey II. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1128–1134.

Wrobel, S. (1993). On the Proper Definition of Minimality in Specialization and Theory Revision. In *Machine Learning: ECML-93, European Conference on Machine Learning*, volume 667 of *LNCS*, pages 65–82. Springer.

Wrobel, S. (1994). Concept Formation During Interactive Theory Revision. 14:169–191.

Wrobel, S. (1996). First-order theory refinement. In De Raedt, L., editor, *Advances in Inductive Logic Programming*, pages 14–33. IOS Press.

Yamamoto, A. (1997). Which Hypotheses Can Be Found with Inverse Entailment? In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNCS*, pages 296–308. Springer.

Yedidia, J. S., Freeman, W. T., and Weiss, Y. (2000). Generalized Belief Propagation. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS-00)*, pages 689–695.

Železný, F. and Lavrac, N. (2006). Propositionalization-Based Relational Subgroup Discovery with RSD. *Machine Learning*, 62(1-2):33–63.

Železný, F., Srinivasan, A., and Page, D. (2002). Lattice-Search Runtime Distributions May Be Heavy-Tailed. In *Proceedings of the Twelth International Conference on Inductive Logic Programming (ILP-02)*, volume 2583 of *LNAI*, pages 341–358. Springer.

Železný, F., Srinivasan, A., and Page, D. (2004). A Monte Carlo Study of Randomised Restarted Search in ILP. In *Proceedings of the 14th International*

Conference on Inductive Logic Prgramming (ILP-04), volume 3194 of *LNAI*, pages 341–358. Springer.

Železný, F., Srinivasan, A., and Page, D. (2006). Randomised restarted search in ILP. *Machine Learning*, 64(1-3):183–208.

Zelle, J. M., Mooney, R. J., and Konvisser, J. B. (1994). Combining Top-down and Bottom-up Techniques in Inductive Logic Programming. In *Proceedings of the E11th International Conference on Machine Learning (ICML-94)*, pages 343–351.

Zhang, N. L. and Poole, D. (1996). Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research*, 5:301–328.

Index

Bayes Ball algorithm, 168
Bayesian Logic Program, 180
Bayesian networks, 162
Bayesian Revision Points, 204
Bottom clause, 18

Combining rule, 180
combining rules, 180
cutoff, 104

d-separation, 165

EM algorithm, 175
Example × Instance in PFORTE, 192
Extensional ILP, 17

First Improvement strategy, 102
FORTE, 27
FORTE examples representation, 28
FORTE learning setting, 28
FORTE Revision operators, 31
FORTE_MBC, 47

Generative and Discriminative training, 173
Greedy Stochastic Hill Climbing strategy, 102
GSAT, 106

ILP, 14
Input variable, 20
Intentional ILP, 17

Learning from entailment, 17
learning from interpretations, 17
Learning from satisfiability, 17
Likelihood, 173
Logic programming terms, 15

Markov networks, 161
MIS culprit clauses, 25
Mode declaration, 20
Multi-Relational Data Mining, 14

Output variable, 20

PFORTE classification, 193
Probabilistic Graphical Models, 161
Probabilistic Inductive Logic Programming, 159
Probabilistic Iterative Improvement, 103
Probabilistic Logic Learning, 159
Propositionalization, 113

Randomised Iterative Improvement, 102

Rapid Random Restarts, 104
Revision operators, 26
Revision points, 25

Simulated Annealing, 103
SLS in ILP, 109
Statistical Relational Learning, 159
Stochastic Hill Climbing strategy, 101
support network, 181

Transfer Learning, 68
Two-levels combining rules, 187

v-structure, 166

WalkSAT, 107

Appendix A

The Laws of International Chess

Here we summarize the main rules of the traditional game of chess, set up by Fédération Internationale des Échecs (FIDE), also known as the World Chess Organisation (FIDE,).

1. Chess is a game played by two people on a chessboard divided into 64 squares of alternating colour, with 32 pieces (16 for each player) of six types. Each player has control of one of two sets of coloured pieces (white and black). White moves first and the players alternate moves. Each type of piece moves in a distinct way. The goal of the game is to checkmate, i.e. to threaten the opponent's king with inevitable capture.

2. At the beginning of the game the pieces are arranged as shown in Figure A.1.

3. Each square of the chessboard is identified with a unique pair of a letter and a number. The vertical *files* are labelled a through h. Similarly, the horizontal *ranks* are numbered from 1 to 8. Each square of the board is uniquely identified by its file letter and rank number. The white king, for example, starts the game on square $e1$.

4. During the game, a piece may be *captured* and then removed from the game and may not be returned to play for the remainder of the game. The king can be put in check but cannot be captured.

5. Pieces can move as follows:

- The king can move exactly one square horizontally, vertically, or diagonally. At most once in every game, each king is allowed to make a special move, known as castling (see below).
- The rook moves any number of vacant squares vertically or horizontally. It also is moved while castling.
- The bishop moves any number of vacant squares in any diagonal direction.
- The queen can move any number of vacant squares diagonally, horizontally, or vertically.
- The knight moves in an "L" or "7" shape (possibly inverted): it moves two squares like the rook and then one square perpendicular to that.
- A pawn can move forward one square, if that square is unoccupied. If it has not yet moved, the pawn has the option of moving two squares forward provided both squares in front of the pawn are unoccupied. A pawn cannot move backward. Pawns are the only pieces that capture differently from how they move. They can capture an enemy piece on either of the two spaces adjacent to the space in front of them (i.e., the two squares diagonally in front of them) but cannot move to these spaces if they are vacant.

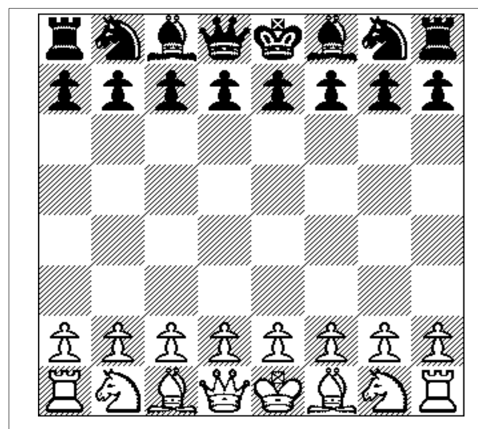

Figure A.1: Initial position of a Chess board: first row: rook, knight, bishop, queen, king, bishop, knight, and rook; second row: pawns

6. *Castling* consists of moving the king two squares towards a rook on the player's

first rank, then moving the rook onto the square over which the king crossed. Castling can only be done if the king has never moved, the rook involved has never moved, the squares between the king and the rook involved are not occupied, the king is not in check, and the king does not cross over or end on a square in which it would be in check (see figure A.2).

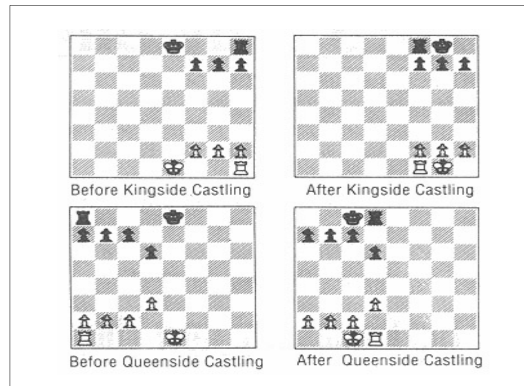

Figure A.2: Castling from the queen side and from the king side (figure is due to *http://www.pressmantoy.com/instructions/instruct_chess.html*)

7. *En-passant* happens if player A's pawn moves forward two squares and player B has a pawn on its fifth rank on an adjacent file and B's pawn capture A's pawn as if A's pawn had only moved one square. This capture can only be made on the immediately subsequent move. In Figure A.3 the black pawn moves from b7 to b5 and the white at c5 capture it en passant, ending up on b6.

8. If a pawn advances to its eighth rank it is *promoted* to a queen, rook, bishop, or knight of the same colour, according to the players desire.

9. When a player makes a move that threatens the opposing king with capture, the king is said to be in check. The definition of check is that one or more opposing pieces could theoretically capture the king on the next move (although the king is never actually captured). If a player's king is in check then the player must make a move that eliminates the threat(s) of capture. The possible ways to remove the threat of capture are: (1) Move the king to a square where it is not threatened. (2)Capture the threatening piece (possibly with

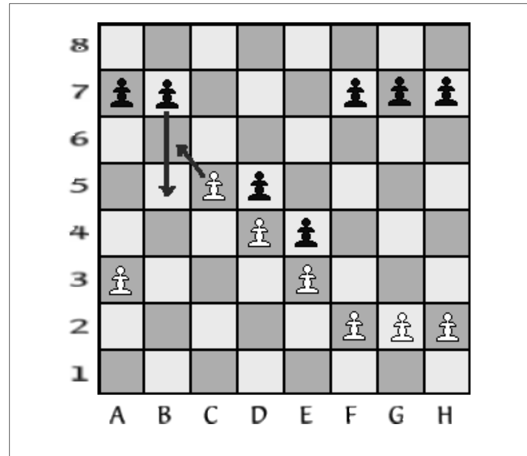

Figure A.3: En-passant move of a pawn in the Game of Chess. Figure is from *http://www.learnthat.com/courses/fun/chess/beginrules15.shtml*

the king, if doing so does not put the king in check). (3) Place a piece between the king and the opponent's threatening piece.

10. If the king is in double check, i.e., threatened by two different pieces, the only piece allowed to move is the king itself, since no other piece is able to take away the king from the threat.

11. If a piece is protecting the king from a check it is only allowed to move to another position where it continues protecting the king (the piece is called as *absolute pin*).

12. If a player's king is placed in check and there is no legal move that player can make to escape check, then the king is said to be checkmated and the game ends.