

Imperial College of Science, Technology and Medicine
Department of Computing

An Artificial Intelligence Framework for Investigative Reasoning

Ramin Ramezani

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College of Science, Technology and Medicine,
July 2014

Abstract

Problem solving is one of the major paradigms in Artificial Intelligence research in which an intelligent task to automate is interpreted as a series of problems to be solved. Various problem solving techniques have been spawned in the field of AI, mostly by concentrating on a certain reasoning approach to tackle a particular class of problems. For instance, theorem proving, constraint solving and machine learning provide powerful techniques for solving AI problems. In all these approaches, background knowledge needs to be provided, from which the system will infer new knowledge. Often, however, in real world scenarios, there may not be enough background information for any single solver to solve the problem. In these situations, some researches have demonstrated the benefits of using *combined reasoning*, i.e., a reasoning process which employs various, often disparate, problem solving techniques in concert, in order to solve a given task. The systems that engage such reasoning processes are called *combined reasoning systems*. Their power draws upon disparate techniques they employ. As such, combined reasoning systems are supposed to be more capable than their constituents. In this thesis we mainly focus on using a combined reasoning approach in solving a type of problems that cannot be solved by any of the aforementioned standalone systems. We refer to this type as *investigation problem* which models to some extent a generic situation which might arise in, say, medical diagnosis or the solving of a crime. That is, there are a number of possible diagnoses/suspects (*candidates*), and the problem is to use the facts of the case to rank them in terms of their likelihood of being the cause of the illness/guilty of the crime. Such ranking often leads to further medical tests/police enquiries focusing on the most likely candidates, which will bring to light further information about the current case. We use the term *dynamic investigation problems* to describe a series of such problems to be solved. Solving each problem entails using the facts of the case, coupled with prior knowledge about the domain to narrow down the candidates to just one. However, when there is no upright solution due to lack of some essential information, additional relevant information can often be found in related past cases thereby irregularities can be observed and utilized. Hence, dynamic investigation problems are hybrid machine-learning/constraint solving problems, and as such are more realistic and of interest to the wider AI community. In this thesis we focus on formal definition, exploration, generation and solution of “*Dynamic Investigation Problems*”, and we develop a framework which performs “*Investigative Reasoning*”, that is a framework in which a combination of reasoning techniques are incorporated in order to tackle dynamic investigation problems.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Professor Simon Colton, for his enthusiasm, support and insightful comments throughout this work. It has been a unique experience to be guided and inspired by a master in the field.

My special thanks to EPSRC and Imperial College London, Department of Computing, for generously funding me with PhD studentship under the grant EP/F036647/1 and also with the departmental scholarship to work on computational models of axiom formation and reformulation in AI problem solving.

Being a member of Computing Department has been an extraordinary experience, surrounded by the most helpful and kind members of staff one could ever imagine. I take this chance to extend my thanks to all of them. I wish my time in this wonderful place would have never finished, but *c'est la vie*.

My deepest appreciation and gratitude to Dr. Amani El-Kholy for her successive supports, encouragements and charismatic presence. Many thanks to Professor Marek Sergot for his guidance, specially through the tough period of my work. I would also like to thank Professor Stephen Muggleton; it was a brilliant opportunity to be able to share ideas with his unique research group and his brilliant contributions in the field of ILP paved my way in understanding many of the concepts reflected in this thesis.

Many thanks to my dear friends and colleagues at Computational Creativity Group and Computational Bioinformatics Group: Alison Pease, Daniel Ramirez Cano, Jeremy Gow, John Charnley, Dianhuan Lin, José Santos, Alireza Tamaddoni-Nezhad, Jianzhong Chen, Robin Baumgarten, Michael Cook, Niels Pahlavi, Pedro Torres, Kai Sun, Zena Hera, Kate Reed, Flaminia Cavallo and Aline Paes. I have had the privilege of spending time with some of the brightest minds I have ever met in my life. Over the past few years, I very much enjoyed sharing ideas and talking to you all. My special thanks to Dr. Alireza Tamaddoni-Nezhad and Dr. José Santos for their brilliant suggestions and comments throughout my project. I should also acknowledge Dr. Daniel Ramirez Cano who has been a true friend and support when the work looked so far-fetched. Daniel, I am so honored to have a friend like you.

Part of the implementation and simulation of this work heavily relied on using departmental

resources. I would like to extend my gratitude to Computing Support Group and specially Dr. Lloyd Kamara for being always supportive. To Lloyd, I learned a lot from you and I am very grateful for all your help.

I feel truly honored to have had the opportunity of being a member of Imperial College. This university with its nurturing environment has always made me feel at home and is certainly my home. My life at Imperial has left an indelible mark on my memory and I will always cherish the time I spent in its exciting atmosphere.

And finally, my uttermost thanks to my parents and my sister, Bahareh, for all their love and unbelievable support. It was certainly not possible without you and I cannot fully express my gratitude to you.

Statement of Originality and Publications

I declare that this thesis was composed by myself, and that the work it presents is my own except where otherwise stated. Some aspects of the conducted work have been covered in the following publications:

- Part of the preliminary experiments, described in chapter 3, was designed to test the feasibility of a combined reasoning approach to a whodunnit problem known as Aunt Agatha. It was presented in the *Automated Reasoning Workshop* in 2009 [107].
- The formal definition and generation of dynamic investigation problems, described in chapters 3 and 4, was partly presented in *Automated Reasoning Workshop, Bridging the Gap between Theory and Practice* in 2010 [106].
- Further experiments on using a combined reasoning technique in constraint solving, described in chapter 6, and investigation on the potential for applying Lakatos-inspired constraint solving were published as a book chapter in *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles*, 2010 [99].
- The argument and relevant experiments that the visual and analogical representations of mathematical concepts can be used by HR automated theory formation system to develop further mathematical concepts and conjectures were published in *Model-Based Reasoning in Science and Technology, Springer 2010* [97].
- The GH system developed and described throughout this system was the subject of a study on creative systems and we investigated the possibility of GH acting as a *serendipitous* system. The discussion was published and presented at the *Fourth International Conference on Computational Creativity* in 2013 [96].
- The experiments and results on dynamic investigation problems, described in chapters 8 and 9, were partly published and presented at *50th annual convention of the Artificial Intelligence and the Simulation of Behaviour (AISB-50), proceedings on Computational Scientific Discovery* in 2014 [36].

© The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

To my family for their endless love and support

Contents

Abstract	i
Acknowledgements	iii
1 Introduction	1
1.1 Objectives and Main Contributions	3
1.2 Thesis outline	5
2 Background - AI Problem Solving Techniques	8
2.1 Introduction	8
2.2 Knowledge Representation and Logical Systems	9
2.2.1 Propositional Logic	10
2.2.2 First-Order Logic	12
2.3 Logic Programming and Prolog	14
2.4 Automated Theorem Proving	15
2.4.1 OTTER	16
2.5 Constraint Solving	17
2.5.1 Formal definition of a CSP	18

2.5.2	Expressing Problems as CSPs	21
2.6	The HR Discovery System	23
2.6.1	Concept Formation	24
2.6.2	Inventing Concepts	25
2.6.3	Making Conjectures	26
2.6.4	Setting up HR	27
2.6.5	HR3	29
2.7	Machine Learning	30
2.7.1	Data mining and Association Rule Mining	33
2.7.2	Apriori Algorithm	35
2.7.3	Weka	38
2.8	Logical and Relational Learning	38
2.8.1	Inductive Logic Programming	39
2.8.2	The normal ILP setting	41
2.8.3	Defining and Bounding the Hypothesis Search Space	43
2.8.4	Inverse Entailment and Progol	46
2.9	Stochastic Logic Programming	48
2.10	Conclusions	49
3	Dynamic Investigation Problems	51
3.1	Introduction	51
3.2	Motivations and Inspirations	52
3.3	The Aunt Agatha Problem	54

3.3.1	Aunt Agatha – Machine Learning	55
3.3.2	Aunt Agatha – Constraint Solving	56
3.3.3	Aunt Agatha – Automated Theorem Proving	57
3.3.4	Mutilated Aunt Agatha	58
3.4	Investigation Problem (IP) - Formal Definition	59
3.4.1	IP: Current Case	60
3.4.2	IP: Past Cases	63
3.4.3	Definition of an IP	65
3.4.4	Isomorphic IP	66
3.5	Definition of a Dynamic Investigation Problem (DIP)	67
3.6	DIP: a toy example	68
3.7	Conclusions	73
4	Automatic Generation of Dynamic Investigation Problems	75
4.1	The GH System – Introduction	75
4.2	Representation language	76
4.3	Automatic generation of Cluedo – style Problems	77
4.3.1	Example of Cluedo Generation	80
4.4	DIP Automatic Generation – Generic Type	81
4.4.1	Generic DIP – Current Case	82
4.4.2	Generic DIP – Past Cases	87
4.5	Translation Suite – TS	88
4.5.1	Translation to Prolog	89

4.5.2	Translation to HR	92
4.5.3	Translation to Weka – Data Flattening for Weka	93
4.6	Conclusions	98
5	GH - The Reality Factors	101
5.1	Motivations	101
5.2	Reality Factor: Weighted Predicates and Suspects	102
5.2.1	Dealing with Reality Issues	105
5.2.2	Example of Generation Based on Weighting	112
5.3	Reality Factor - Negation	116
5.4	Conclusions	120
6	Solving Preliminary Case Studies	121
6.1	Introduction	121
6.2	Aunt Agatha Problem	122
6.2.1	Solving the Aunt Agatha Problem: HR and OTTER	123
6.2.2	Solving the Aunt Agatha Problem: SICStus Prolog	123
6.2.3	Solving the Aunt Agatha Problem: Progol and Prolog	125
6.2.4	Solving the Aunt Agatha Problem: HR and CLPFD	126
6.3	Cluedo-style Problems	129
6.3.1	Solving Cluedo-style problems: HR	129
6.3.2	Solving Cluedo: HR and Partial Solver	133
6.3.3	Dealing with different answers	135
6.4	Conclusions	138

7	A Combined Reasoning DIP Solver	140
7.1	Introduction	140
7.2	GH – A Combined Reasoning System	141
7.3	Solving DIPs	145
7.3.1	GH employing HR2	148
7.3.2	GH employing Weka	151
7.3.3	GH employing Progol	156
7.3.4	GH employing Progol +	159
7.4	GH employing <i>Pe-Pl</i> and Reality Factor	161
7.4.1	Partial Solver	164
7.5	Conclusions	165
8	Experimentations and Configurations	166
8.1	Introduction	166
8.2	Calculating the Error Rate of a Solution	167
8.3	DIP types	169
8.3.1	Level of difficulty: Type 1	170
8.3.2	Level of difficulty: Type 2	171
8.3.3	Level of difficulty: Type 3	172
8.4	Dynamic aspect and Temporal Solutions	174
8.5	Experiments	175
8.6	An example of the results	180
8.7	Configurations and Implementation Details	181

8.8	Conclusions	182
9	Results and Analysis	183
9.1	Introduction	183
9.2	Experimental Results	184
9.2.1	Type-1 DIPs: Multiple Rules	185
9.2.2	Type-2 DIPs: Single Rule	188
9.2.3	A note on Progol+	191
9.2.4	Type-3 DIPs: Reality Factors	193
9.2.5	Temporal Solutions	194
9.2.6	A note on Occam's Razor	196
9.2.7	Rule Evaluation Measures in Association Rule Mining	198
9.3	Commentary on the Results	201
9.4	Conclusions	204
10	Conclusions	206
10.1	Summary of the Thesis	207
10.2	Future directions	209
10.2.1	Argumentation	209
10.2.2	Theory Revision	210
10.2.3	A cognitive approach for solving DIPs	211
A	Generation Appendix	215
A.1	Example of Cluedo Past Cases	215

A.2	Generic DIP: A set of past cases	216
A.3	Translation Suite: DIP past cases in Progol format	217
A.4	Translation Suite: DIP past cases in HR format	219
A.5	Translation Suite: DIP past cases in Weka format	222
B	Solution Appendix	224
B.1	Solving Aunt Agatha Problem	224
B.1.1	Aunt Agatha: OTTER input	224
B.1.2	Aunt Agatha: OTTER output	225
B.1.3	Aunt Agatha: CLPFD Translation	228
B.2	Cluedo: CLPFD Translation and Solution	231
	Bibliography	234

List of Tables

2.1	Examples of logical formulae in propositional logic	11
2.2	Examples of rewriting rules	11
2.3	Examples of propositional inference rules	12
2.4	First-order logic inference rules targeting quantifiers	13
2.5	Some of Weka parameters for Apriori algorithm [5]	39
3.1	Answer: $T = t_1$, rules: C_1, C_2	70
3.2	Answer: $T = t_1$, rules: C_1, C_2, C_3	71
3.3	Answer: $T = t_2$, rules: C_1, C_2, C_3, C_4	71
3.4	Answer: $T = t_2$, rules: C_1, C_2, C_3, C_4, C_5	72
6.1	A subset of solutions	136
6.2	Cluedo solutions – different rules & times	137
7.1	HR Concept formation	150
8.1	Summary of experiments	179
9.1	Execution times in milliseconds for Weka, HR3, Progol and HR2, running on a 3.2GHz processor, averaged over 1000 Type-1 investigation problems requiring solutions of the form $Lx Ay$ (with exactly x literals of arity at most y	185

9.2	Error Rates for Weka, HR3 and Progol, running on 3.2GHz processor, averaged over 1000 Type-1 investigation problems (number of <i>target candidates</i> > 1) requiring solutions of the form LxAy. Significant execution times in milliseconds are given in brackets.	188
9.3	Percentage error rates of Progol averaged over 1000 Type-2 investigation problems requiring solutions of the form LxAy. The ratio of correct solutions to the total number of generated solutions is given in brackets. Progol parameters: i=5, nodes=1000	189
9.4	Execution time in milliseconds, running on a 3.2GHz processor, averaged over 1000 Type-2 investigation problems requiring solutions of the form LxA1. Progol parameters: nodes=200, i=1, set(posonly).	192
9.5	Percentage error rates of Weka and Progol averaged over 1000 Type-3 investigation problems requiring solutions of the form LxAy. Execution times in milliseconds on a 3.2GHz processor are given in brackets.	193

List of Figures

2.1	A possible solution to the 8-queen puzzle [127]	17
2.2	Making arc consistency in both directions (x, y) and (y, x) [118]	19
2.3	Example of constraint interpretation	22
2.4	Example of a concept definition and the examples satisfying the concept	24
2.5	HR data-table for the concept of murderer	25
2.6	Inverse resolution rule in first-order logic	33
2.7	Michalski train problem	45
2.8	Michalski's train problem: mode declarations	45
2.9	Michalski's train problem: positive and negative examples	46
3.1	Aunt Agatha: part of a case study	56
3.2	Part of Aunt Agatha problem in Prolog syntax	57
3.3	Part of the input to OTTER	58
3.4	Conditions of $l - connectedness$	59
3.5	Mapping function $M : CurrentCase \longrightarrow PastCases$	66
4.1	Cluedo generation algorithm	79
4.2	Example of a Cluedo scenario - current case	81

4.3	DIP generation: current case input parameters	82
4.4	DIP generation: current case – part 1	84
4.5	DIP generation: current case – part 2	85
4.6	Generic DIP – part of the current Case	87
4.7	TS partial translation to HR Domain File	93
4.8	Example of Weka input	94
4.9	Weka format – data segment representing Polio	96
4.10	Flattening DIP past cases for Weka	97
4.11	Part of a DIP translated to Weka	98
4.12	The GH System GUI	100
5.1	Siberia problem in GH format	103
5.2	Occurrence of suspects in the important predicates	113
5.3	Ranking of suspects	113
5.4	Part of a current case	114
5.5	Ordering of suspects	114
5.6	Example of a past case based on rankings	116
5.7	Past case generation based on partial current case ranking	117
5.8	Rank of suspects based on $T = 2$	117
5.9	Partial Prolog code – negated queries	119
6.1	A subset of HR output	124
6.2	Partial background knowledge from Aunt Agatha problem	124
6.3	The murderer and the victim are the same	126

6.4	The murderer and the victim are <i>not</i> the same	126
6.5	Part of the automatic translation of the Aunt Agatha problem to CLPFD	127
6.6	Aunt Agatha CLPFD solution	128
6.7	A Cluedo scenario	131
6.8	A part of HR conjectures	132
6.9	Partial Prolog code for TS – negated queries	133
7.1	Schematic diagram of the solution process in GH	143
7.2	The set of answers: ‘A’	147
7.3	<i>type-1</i> DIP facts and rules	149
7.4	Examples of HR concepts	149
7.5	Data table for concepts:	150
7.6	DIP partial scenario and HR concept	151
7.7	A subset of Weka extracted rules	155
7.8	A subset of the TS translation of Weka generated rules	156
7.9	An example of mode declarations in Progol	159
7.10	Part of Progol+ output	160
7.11	An example of an SLP adapted from [40]	161
7.12	Input to <i>Pe-Pl</i>	163
7.13	Correctly selected Hypothesis	164
8.1	‘A’ representing the set of solutions	167
8.2	Comparing the GH solution with the embedded answer.	168
8.3	Attribute of the ruleset	170

8.4	An example of an embedded ruleset – Type-1 DIP	171
8.5	GH result using the reality factors	173
8.6	Distribution of information in dynamic experiments	175
8.7	An overview of dynamic process (generation and solution) in GH	176
8.8	The header of a file that stores the error rates	177
8.9	Part of the storage file	180
8.10	Part of the results summary	181
9.1	Error rates in type-1 DIP (Zero error rates for HR3 and Weka)	186
9.2	Distribution of Progol Error Rate - Type-2 DIPs	190
9.3	Temporal Error Rates in Weka and Progol	195
9.4	Percentage error rates of various interestingness measures, averaged over 1000 Type-3 investigation problems requiring solutions of the form LxAy	199

List of Algorithms

4.1	Cluedo Generation	79
4.2	DIP Generation: current case – part 1	84
4.3	DIP Generation: Current Case – part 2	85
4.4	Flattening DIP past cases for Weka	97

Chapter 1

Introduction

Problem solving is one of the major paradigms in Artificial Intelligence research in which an intelligent task to automate is interpreted as a series of problems to be solved. Various problem solving techniques have been spawned in the field of AI, mostly by concentrating on a certain reasoning approach to tackle a particular class of problems. For instance, theorem proving, constraint solving and machine learning provide powerful techniques for solving AI problems. In all these approaches, background knowledge needs to be provided, from which the system will infer new knowledge.

Often, however, in real world scenarios, there may not be enough background information for any single solver to solve the problem. In these situations, some researches have demonstrated the benefits of using *combined reasoning*, i.e., a reasoning process which employs various, often disparate, problem solving techniques in concert, in order to solve a given task. The systems that engage such reasoning processes are called *combined reasoning systems*. Their power draws upon disparate techniques they employ. As such, combined reasoning systems are supposed to be more capable than their constituents. They are also expected to be more effective than the stand-alone systems in which only one form of reasoning is harnessed.

In this thesis we mainly focus on using a combined reasoning approach in solving a type of problem that cannot be solved by any of the aforementioned standalone systems. We refer to this type of problem as *investigation problem* which models to some extent a generic situation

which might arise in, say, medical diagnosis or the solving of a crime. That is, there are a number of possible diagnoses/suspects (*candidates*), and the problem is to use the facts of the case to rank them in terms of their likelihood of being the cause of the illness/guilty of the crime. Such ranking often leads to further medical tests/police enquiries focusing on the most likely candidates, which will bring to light further information about the current case. Hence, we use the term *dynamic investigation problems* to describe a series of such problems to be solved.

Solving each problem entails using the facts of the case, coupled with prior knowledge about the domain to narrow down the candidates to just one. Hence, a natural way to model such problems is as a constraint satisfaction problem (CSP), with one variable which takes one of n values, each representing a candidate, and the facts of the case acting as the constraints. Solving the case means finding a value to assign to the variable which does not break the constraints. Often, however, not all the essential information is readily available, hence these problems are best modeled as partial CSPs. As such, especially during the early stages of the investigation, there will be no outright solution, and the constraints in the CSP need to be used to rank the candidates for further investigation. Additional relevant information can often be found in related past cases, from which regularities can be observed and utilized, and consultation of previous case studies is part of the investigation process.

Using only the data about the current case, these problems can be modeled most easily as a dynamic constraint satisfaction problem (DCSP) which has a single solution. However, this does not take into account the case studies, within which the pertinent information about the current case may be hidden. Using only the case studies, these problems can be modeled most easily as a machine learning problem, from which the profile of the guilty party can be learned. The profile can then be applied to the current suspects, and the one most fitting the bill is the guilty one. Hence dynamic investigation problems are hybrid machine-learning/constraint-solving problems, and as such are more realistic and of interest to the wider AI community.

In this thesis we focus on formal definition, exploration, generation and solution of “*Dynamic Investigation Problems*”. We also develop a framework which performs “*Investigative Reason-*

ing”, that is a framework in which a combination of reasoning techniques are incorporated in order to tackle dynamic investigation problems. In summary, this research is motivated by the situations that occur in criminal or medical investigations, and is further stimulated by the desire to build an efficient framework in which a combination of reasoning techniques can be employed to solve such problems.

1.1 Objectives and Main Contributions

The aim of this thesis is to formally define, design and solve problems which to some extent model real-life situations where a guilty party has to be chosen from a number of suspects. The decision depends on a changing (dynamic) set of facts and constraints about the current case and a changing set of case studies of a similar nature to the current case. Such situations occur in criminal or medical investigations and we name them as *investigation problems*. In order to solve such problems, this research aims to build a framework in which a combination of disparate reasoning techniques are employed, in concert, in order to solve investigation problems, i.e., a framework for *investigative reasoning*. Finding the best practice of investigative reasoning is in-line with the broader object of our work which is to advance the problem solving paradigm in AI research.

This research resembles a journey through creating and solving medical or criminal mysteries. The itinerary of this journey starts by performing a combination of reasoning techniques on a simple investigation problem. The problem is defined in such a way that some pieces of information, crucial to solving the problem, are intentionally removed. In addition, similar case studies are designed to hold extra constraints for the dynamic constraint satisfaction problem (DCSP) of the current case. Extracting the extra constraints is not a straightforward machine learning exercise, as the data is unlabeled. Therefore, HR descriptive machine learning system is employed by which the information from the case studies are mined. However, the HR system also produces a number of conjectures about the data which are empirically true, but of no value to the DCSP. Hence, the conjectures are filtered from HR and turned into

usable additional constraints. Subsequently, these constraints are tested to find whether they can help solving the investigation problems. This preliminary experiment paves the way to have a better understanding of how a dynamic investigation problem should model such real-world scenarios. The journey continues by formalizing dynamic investigation problems and furthermore, implementing methods to systematically generate such problems. The final stop of the journey elaborates on combining various techniques to solve dynamic investigation problems and evaluating the solution procedure. The main contributions are described as follows:

- Dynamic investigation problems were formally defined in terms of existing terminology relating to machine learning and constraint solving. Such problems were designed specifically to be more realistic than the kind of problems solvable by standalone Artificial Intelligence approaches, such as theorem proving, machine learning or constraint solving. The problems contain facts and rules about a current investigation case and many other cases than can bear resemblance to the current case. Similar to real scenarios, the suspects, facts and/or rules of the investigation case can change at different times, hence the dynamic nature of the problem. The aim is to correctly rule out the bystanders and to identify the guilty party in the light of changing information.
- The GH system was built which automatically generates dynamic investigation problems and was experimented with its usage to produce more varied problem sets in terms of difficulty. The problems were generated in first-order logic. A translator was developed that can automatically translate the problems from first-order logic into the syntax of Progol, HR and OTTER. An algorithm was developed that can automatically take dynamic investigation problems in first-order logic and represent them in attribute-value format of Weka.
- A set of factors were developed to assign data with hierarchical importance. These factors, named as *reality factors*, were taken into account to modify the generation process thereby producing more difficult scenarios. Taking such factors into account is a stochastic approach for prioritizing the information. A set of formulae was introduced for ranking and weighting the predicates and suspects. These formulae were used to produce more

realistic problems and hence harder to solve.

- A framework was developed by incorporating Progol, Weka, HR, Pe-Pl and SICStus CLPFD along with a partial solver to tackle DIPs. A thorough comparison and evaluation was made between GH's embodied systems and downgrading/upgrading technique in logical and relational learning.

1.2 Thesis outline

Chapters 1 and 2 provide the background material and describe the foundations of our work:

- Chapter 1: **Introduction**

I briefly describe the objectives, initial case-study, motivations and contributions.

- Chapter 2: **Background – AI Problem Solving Techniques**

I discuss the problem solving techniques that have been used in my research. Laying the foundations, I briefly provide background to logical systems, logical and relational learning, constraint solving, machine learning and automated theorem proving.

Chapters 3, 4 and 5 are dedicated to formal definition and generation of the problems I aim to tackle:

- Chapter 3: **Dynamic Investigation Problems**

I formally define investigation problems and dynamic investigation problems and provide few case studies that initially inspired me.

- Chapter 4: **Automatic Generation of Dynamic Investigation Problems**

This chapter is dedicated to defining and discussing the algorithms I developed for generating dynamic investigation problems. I also describe the algorithms I developed for translating DIPs.

- Chapter 5: **The Reality Factors**

I discuss a set of factors that I developed to make the problems more realistic. These factors divide the background information in terms of importance into high and low.

I complete the development of GH system, a framework for investigative reasoning, and discuss the solution mechanisms:

- Chapter 6: **Solving Preliminary Case Studies**

I discuss the solution methodologies employed in tackling the preliminary case studies. Solving those case studies were among the first attempts at investigating ill-formed problems that neither of the standalone AI techniques such as constraint solving, machine learning or automated theorem proving could solve.

- Chapter 7: **A Combined Reasoning DIP Solver**

I formalize what I mean by finding a solution to a DIP and discuss how GH proceeds when it cannot find a *single* or a set of *definite* answers. I will demonstrate what I mean by a solution and how GH infers a set of partial solutions with various degrees of possibilities. This chapter will elucidate how GH utilizes a combination of AI systems to tackle DIPs.

I investigate the efficiency and performance of the GH system:

- Chapter 8: **Experimentations and Configurations**

I discuss some of the configuration and implementation details of GH required to guarantee a smooth interaction between the bespoke systems. I also discuss how GH is flexible to a set of input/output syntaxes.

- Chapter 9: **Results**

I illustrate and discuss the results of the experiments. I also provide a detailed discussion in comparing the various reasoning systems employed by GH.

Finally, I draw conclusions of the thesis and describe the future plans for taking this research further.

- Chapter 10: **Conclusions**

I summarize the research and draw conclusions. I also provide some applications to the GH system and describe the further steps that should be taken in order to make GH more applicable in real worlds scenarios.

Chapter 2

Background - AI Problem Solving Techniques

2.1 Introduction

One of the major paradigms in Artificial Intelligence research is *problem solving* in which an intelligent task to automate is interpreted as a series of problems to be solved [37]. To fulfill the intelligent task, an intelligent agent is designed in such a way that it can perceive the information about a problem and perform certain actions to solve it. Accessing and utilizing the information is therefore essential for an intelligent agent in order to act rationally. The way information is often presented to and generated by an intelligent agent, to a large extent, depends on the type of problem we are addressing. This chapter provides some background material to the subject of our project, where we investigate solving a type of AI problems that we will elaborate on throughout this thesis. Those problems will be introduced and formally defined as *dynamic investigation problems* (DIP) in chapter 3. To cater for those problems, in this chapter, we will discuss knowledge representation and first order logic whereby our problems will be represented, and ultimately a logic-based AI system will be designed to solve it.

A logic-based AI system should benefit from reasoning techniques in order to solve an AI

problem. Reasoning, in general, is the ability to *infer*, and *automated reasoning* is concerned with building software that can enable an AI agent to completely or almost completely reason autonomously [8, 52, 103]. Reasoning can be done in different ways such as deduction, induction and abduction. The term “automated reasoning” has largely been used to describe a sub-field of itself concerned with deduction known as Automated Theorem Proving (ATP). However, the overall goal of automated reasoning is to mechanize various reasoning techniques which we will discuss in this chapter. Therefore, we define an automated reasoning system as a system that (a) provides new information, given the background knowledge, through the process of *logical reasoning* and (b) has a logical representation scheme by which the information is described.

In chapters 6 and 7 we will discuss a framework for “*combined reasoning*”. The phrase combined reasoning refers to a system which harnesses various, often disparate, AI reasoning techniques and incorporates them into a single whole in order to solve a problem. As we have not yet discussed our work, in this chapter we will only provide general background for making such frameworks and focus on discussing the reasoning techniques.

2.2 Knowledge Representation and Logical Systems

One of the major design aspects of an AI system involves delineation of the problem class (*problem domain*) which an AI agent is required to tackle. An agent needs to grasp the information about the environment and infer based on that information in order to find the solution. Therefore it is crucial for an agent to have proper knowledge about the environment and appropriate reasoning techniques. Knowledge can be either observable and easily representable by certain rules or statements, or it can be achieved through forms of experiences that can not be easily described. Reasoning, on the other hand, will help the system to overcome the knowledge of partially observable environments and hence, enables the agent to infer hidden aspects of the current knowledge. Knowledge-Based agents can learn new knowledge about the environment and can ultimately adapt themselves to environmental changes. According to [112], “The central component of a knowledge-based agent is its knowledge base or KB”. A knowledge base is

a set of statements expressed in a specific language called knowledge representation language. Upon every call to the agent, the current perception is delivered to the knowledge base and the question arises about actions that the agent should take. In order to answer the question, extensive inferences takes place about the current situation of the world, possible outcomes of any probable action and so on.

In summary, agents are provided with information which is expressed in some form of knowledge representation language. This language must have the characteristics of a robust language that can avoid any ambiguity about the information and can benefit from certain rules and principles. Logic is a language that can fulfill such requirements and is preferred to natural languages in expressing knowledge because it is a formal language which can represent the knowledge about the objects, and can reason about the properties of them [52]. The rules of forming the logical language are called syntax which varies depending on the language being used. It can be used in various ways; for instance, it can be used to construct proofs and refutations: the system uses the facts (or statements we believe as facts) called *axioms* and a set of deduction rules called inference rules. Therefore, a logical system is a language for describing logical formulae together with a set of inference rules that can derive new logical formulae from old ones. Logical systems, depending on their representational schemes, are divided into various types such as *propositional logic* and *first-order logic*. In this section we briefly cover these two systems along with some of the terminologies they use. Note that throughout this chapter we use Prolog notation, i.e., lower case letters for representing constants and upper case for variables. We will also briefly cover Logic Programming and Prolog in §2.3.

2.2.1 Propositional Logic

Proposition is a declarative statement which is either *true* or *false* such as “*garfield_is_a_cat*”, or “*ai_is_very_interesting*”. Therefore, each proposition has two *models*, i.e., two possible states (true or false). The language of *Propositional Logic* contains *propositions* together with *connectives*. Connectives represents notations in reasoning and are used to form *complex sentences*, i.e., sentences that are constructed from joining propositions, parentheses and connectives. There

are five connectives: (a) *negation* or $\neg$ which means *NOT*, (b) *conjunction* or $\wedge$ which means *AND*, (c) *disjunction* or $\vee$ which means *OR*, (d) *implication* or $\rightarrow$ which means *implies* (if premise then conclusion) and (e) *equivalence* or $\leftrightarrow$ which indicates *if and only if*. In propositional logic, a *well-formed formula (wff)* or simply a *formula*, is a string containing propositions and connectives where the truthfulness of it can be evaluated. The value of wff depends upon the value of the propositions and the connectives being used. Some examples of the wff are illustrated in table 2.1, where the lower case letters represent propositions.

wff	value
$\neg p$	true if p is false and false if p is true
$p \vee q$	true if any of them is true and false if both are false
$p \wedge q$	true if both p and q are true, otherwise false
$\neg p \vee q$	true if q is true or p is false
$p \rightarrow q$	false if p is true and q is false, otherwise true
$p \leftrightarrow q$	true if both have the same value, otherwise false
$p \rightarrow (q \wedge z) \leftrightarrow ((p \rightarrow q) \wedge (p \rightarrow z))$	always true

Table 2.1: Examples of logical formulae in propositional logic

The last formula in the table is an example of a *tautology* where a sentence is true irrespective of the true value of its individual propositions. Tautologies enable us to establish if a set of sentences are having the same meaning. If two sentences are true for exactly the same set of models they are called logically equivalent. *Logical equivalences* enable us to rewrite a sentence into another without changing its meaning. A set of rewriting rules are shown in table 2.2.

property	connectives (if applicable)	example
Commutativity	$\wedge, \vee, \leftrightarrow$	$p \vee q \equiv q \vee p$
Associativity	$\wedge, \vee$	$p \vee (q \vee z) \equiv (p \vee q) \vee z$
Distributivity	$\wedge, \vee, \rightarrow$	$p \vee (q \wedge z) \equiv (p \vee q) \wedge (p \vee z)$
Double Negation	$\neg \neg$	$\neg \neg p \equiv p$
De Morgan's Law	$\wedge, \vee$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Contraposition		$p \rightarrow q \equiv \neg q \rightarrow \neg p$
Replace Implication		$p \rightarrow q \equiv \neg p \vee q$
Replace Equivalence		$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$

Table 2.2: Examples of rewriting rules

Propositional Inference rules enable us to derive new logical formulae given a set of formulae that is assumed to be true. Inference rules are suitable to generate proofs in propositional logic.

Sometimes a system uses collection of inference rules to prove new formulae (conclusions) from the formulae that are taken to be true (premises) [76]. As such, the system may use *direct* routes from premises to reach the proof or sometimes it may use *indirect* inference steps. In an indirect approach, the system assumes that the negation of the conclusion is true and tries to derive (if applicable) a contradiction, $\perp$, to the problem premises. We say that the new formulae are *inferred* from the given formulae. The symbol $\vdash$ is used to indicate the inference, for instance, $\alpha \vdash \beta$ indicates that β is inferred from α . A set of inference rules are given in table 2.3.

Inference Rule	Given	Result
Modus Ponens	$p \rightarrow q, p$	q
And-Elimination	$p_1 \wedge p_2 \wedge \dots \wedge p_n$	$p_i \ (1 \leq i \leq n)$
And-Introduction	$p_1, p_2, \dots, p_n$	$p_1 \wedge p_2 \wedge \dots \wedge p_n$
Or-Introduction	p_i	$p_1 \vee p_2 \vee \dots \vee p_n \ (1 \leq i \leq n)$
Unit Resolution	$p \vee q, \neg q$	p

Table 2.3: Examples of propositional inference rules

2.2.2 First-Order Logic

The problem with the propositional logic is its restriction in expressing “things” and asserting “relations” between things. That is, propositional logic only allows us to represent true or false facts about the world. To overcome the lack of expressiveness, propositional logic has been extended to first-order logic (FOL). First-order logic (or predicate logic) is a formal deductive system that allows us to distinguish between objects (terms) and express a concept (predicate) about objects. An object or a *term* is any variable, constant or function, applied to a term and a *predicate* is a concept expressing a property of an object or stating a relation between various objects. Predicates take on values of true or false. For instance, “Socrates is a philosopher” is expressed in first-order logic as: *philosopher(socrates)*, and, similarly, Plato is Socrates’ student can be stated as: *student(plato, socrates)*. In the above examples, variables were instantiated or *grounded* to the values *socrates* and *plato* which are called *constants*. Predicates and propositions with no connectives are called *atoms*. A *literal* is any predicate or its negation applied to any term. In other words, a literal is an atom or its negation, and a *clause* is a disjunction of literals. A normalized format for representing knowledge base is *Conjunctive Normal Form*.

CNF is in the form of conjunction of clauses where each clause, itself, is a disjunction of literals. For instance, $(p \vee \neg q) \wedge (q \vee z)$. It is possible to translate first-order sentences to CNF by a set of logical manipulations such as renaming variables apart, moving negation signs inwards and removing the quantifiers. Universal quantifiers are eliminated in that all sentences in CNF are assumed to be universally quantified. Existentially quantified variables are replaced by ground terms in a process called *skolemization*.

First-order logic also includes *quantifiers* by which we can express the properties of entire collections of objects. With the use of quantifiers, we can extend our interpretations and be clear about variables in a sentence. By using universal quantifier ($\forall$), we consider all the instantiations of a variable. For example, “all philosophers are humans” can be stated as: $\forall x \text{ philosopher}(x) \rightarrow \text{human}(x)$ which reads “for all x, if x is a philosopher, then x is a human. Existential quantifiers indicate that there can be at least one variable instantiation which satisfies the sentence. For example, “there exists a human that is a philosopher” is stated as: $\exists x \text{ human}(x) \rightarrow \text{philosopher}(x)$ and reads as “there exists an x such that if x is human, then x is a philosopher”.

All the inference rules mentioned in propositional logic can be applied in FOL. In addition, there are few more inference rules that are applied only in FOL, targeting the quantifiers, as described in table 2.4.

Inference Rule	Given	Result
Universal Elimination	$\forall X p(X)$	$p(c)$ where c is a ground term
Universal Introduction	$p(c)$ satisfied for all possible c	$\forall X p(X)$
Existential Elimination	$\exists X p(X)$	$p(c)$ grounded with arbitrary c
Existential Introduction	$p(c)$	$\exists X p(X)$

Table 2.4: First-order logic inference rules targeting quantifiers

For more details on mathematical logic and its history, we recommend [128].

2.3 Logic Programming and Prolog

Amongst various programming models, declarative programming paradigm is the one in which the user does not describe the control flow. That is, given a set of information about the problem, the user only declares what a desired output to a function should look like. It is important for any declarative language to define how the knowledge about the problem is represented and what problem-solving strategies should be employed. Logic programming is a model which embodies declarative programming aspects in which the problem solving algorithm consists of (a) a logic component by which the knowledge is represented and (b) inference rules as control components whereby problem-solving strategies are determined [66]. Logic programming is based on a subset of first-order logic which consists of first-order *horn clauses*. Horn clauses are logic clauses with at most one positive literal. The horn clause with precisely one positive literal is called a *definite clause*. It is often convenient to rewrite and see a definite clause in an implication form. For example, let us consider the following clause: $\neg B_1 \vee \dots \vee \neg B_n \vee H$. Using the equivalence rewriting rule ($\neg A \vee B \equiv A \rightarrow B$) and De Morgan's law, we can rewrite the clause as: $B_1 \wedge \dots \wedge B_n \rightarrow H$. A definite clause can have 0 or more literals ($B_1, \dots, B_n$) as the antecedent and exactly one consequent H . Antecedents are called *Body* literals and the consequent is called *head*.

Prolog is a general-purpose declarative programming language, created in 1972. It implements the logic programming paradigm. There are various commercial and open source compilers for Prolog such as YAP [38] or SWI-Prolog [131], however, SICStus Prolog [50] is the most well-known version that we opted to use in our project. Prolog is mainly used in artificial intelligence applications such as knowledge representation, constraint logic programming, inductive logic programming and natural language processing. The control components of the language is based on a built-in resolution mechanism. Prolog is restricted to use horn clauses [68] as horn clauses allow Selective Linear Definite clause resolution (SLD-resolution) [67]. This refinement of resolution is sound and refutation complete for horn clauses [75]. A problem is represented in Prolog as a set of definite clauses (*rules*) and bodiless clauses (*facts*), and SLD-resolution is used to prove queries by binding the variables to values that can satisfy the query. We will

discuss resolution in the following sections. For detailed overview of Logic Programming, we recommend [75].

2.4 Automated Theorem Proving

Automated theorem proving (ATP) deals with building computer programs that can prove, where possible, that a statement (*conjecture*) can be the *logical consequent* of a set of statements (*axioms* and *hypotheses*). Therefore, given a set of information, it uses the inference rules of logic in order to prove/disprove a theorem. The proof produced by the system describes the process in which the conjecture is logically followed from axioms and hypotheses, rather than just stating the result as an argument. ATP has been notably successful in applications such as mathematics, software creation and software and hardware verification.

The language in which the problem is described for a prover is often first-order logic, although there are ATP systems that can tackle problems represented in higher-order logic. There are two distinctive differences between various ATP systems: (a) the language in which the problem is represented and (b) the reasoning approaches employed by the systems. These characteristics form the underlying strength of various types of ATP systems. OTTER [79], Vampire [108] and Waldmeister [58] are well-known ATP systems at first-order logic level, and Nqthm [16] and Coq [12] are examples of higher order systems. Many of these systems employ *resolution-based* methods as their reasoning mechanism. In addition to inference rules, ATP systems may also benefit from extended facilities such as term rewriting, term orderings and strategies for directing and restricting search mechanisms. For instance, the successor of OTTER, Prover9, uses *paramodulation* which is an improvement in resolution-based method. Paramodulation enables the system to deal with equality predicate. Waldmeister employs completion without failure technique in order to deal with logical formulae including equality [10].

Resolution [111] is a generalized version of the unit resolution inference rule we showed in table 2.3. Resolution is a *sound* operator, i.e., if the following: $c_1 \wedge c_2 \vdash_{res} c$, holds for two clauses c_1 and c_2 , then $c_1 \wedge c_2 \models c$ also holds. However, resolution is not a *complete* operator

in propositional or definite clause logic which are the focus of this section. That is, in case of having two clauses C and c , if $C \models c$, using the *resolution derivation*, we cannot prove that $C \vdash c$. Resolution derivation or *proof* is a sequence of resolution steps that are needed to prove a clause c from a set of clauses $\{c_1, \dots, c_n\}$. Resolution is *refutation* complete. *Proof by refutation* is a popular technique in proving theorems. Using this technique, the agent assumes that the theorem to prove is false. It then tries to prove (if applicable) that this assumption leads to a contradiction to the initial axioms that were initially considered to be true. Refutation can be formulated as: $C \models c$ iff $C \wedge \neg c \models \square$. Proof by refutation is practically a set of resolution steps leading to an empty clause. In summary, let C be a set of clauses, then C is unsatisfiable, $C \models \square$, if and only if there is a resolution derivation (a set of resolution steps) of $\square$ from C . Resolution is an effective procedure in deciding logical entailments due to soundness and refutation completeness. Binary resolution and full resolution are shown as follows:

Binary Resolution

$$\frac{A \vee B, \quad \neg C \vee D}{\text{Subst}(\theta, A \vee D)} \quad \text{Subst}(\theta, B) = \text{Subst}(\theta, C)$$

Full Resolution

$$\frac{P_1 \vee P_2 \vee \dots \vee P_j \vee \dots \vee P_m, \quad Q_1 \vee Q_2 \vee \dots \vee Q_k \vee \dots \vee Q_n}{\text{Subst}(\theta, P_1 \vee \dots \vee P_{j-1} \vee P_{j+1} \vee \dots \vee P_m \vee Q_1 \vee \dots \vee Q_{k-1} \vee Q_{k+1} \vee \dots \vee Q_n)} \\ \text{Unify}(P_j, \neg Q_k) = \theta$$

2.4.1 OTTER

Otter is an automated deduction system for first-order logic with equality [79]. OTTER inference rules are based on resolution and paramodulation with extended facilities such as search directing and restricting. OTTER has been one of the first widely distributed automated theorem provers and has been applied to various areas of mathematics and formal logic. OTTER reads an input file which contains a set of axioms about the problem, the negation of the theorem that is to be proved and some control information. The theorem to prove needs to be negated

due to proof by refutation methodology. After encoding a problem in first order logic, the user chooses the inference rules and sets the control options to control the processing of inferred clauses. For instance, the user specifies the way sentences are going to be parsed. As for the output, OTTER writes the entire search and the proof (if exists) to a pre-specified file. Input to OTTER is represented in first-order logic that can contain equality predicate. In chapter 6 (§6.2) we will describe when and how OTTER was employed in our initial investigations. We will also show an example of input file and generated proof in Appendix B.1.

2.5 Constraint Solving

The study of constraint satisfaction problems (CSP) has been a major subject of research in AI for many years. CSPs appear in many areas such as scheduling, combinatorial problems and vision [130]. A CSP is a problem which consists of a finite set of *variables*, each of which has a finite *domain* of *values* that can be assigned to it, and a set of constraints that impose limits on values that all variables can take simultaneously [127]. In order to solve a problem, values should be *assigned* to variable in such a way that all the constraints are *satisfied*. A very commonly used example of a CSP is *n-queens* problem which is to place n queens in $n \times n$ chess board in such a way that no queen threatens others, i.e., no two queens should be on the same diagonal, row or column. Figure 2.1 illustrates a solution to the 8-queen puzzle. In order to formalize the 8-queen problem as a CSP, 8 variables should choose a value from A to H while satisfying the constraint that no queen should threaten the others.

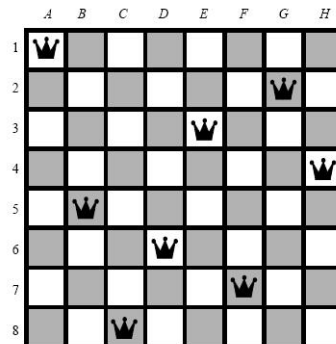


Figure 2.1: A possible solution to the 8-queen puzzle [127]

2.5.1 Formal definition of a CSP

A CSP is composed of:

- Set of *variables*: $X = \{x_1, x_2, \dots, x_n\}$
- *Domain* of the variable consists of a set of possible values that a variable x_i can take and is represented as D_i . The domain may contain *boolean*, *numerical* or *symbolic* values. In each case the variables are called boolean, numerical or symbolic, respectively.
 - *Label* is a variable-value pair (x, v) and indicates the assignment of a value to a variable. The label should be contained in the variable domain, i.e., $v \in D_x$. Compound Label is the simultaneous assignment of value to a set of variables. For instance, assigning $v_1, v_2, \dots, v_n$ to $x_1, x_2, \dots, x_n$, respectively. A compound label can be empty.
- *Constraint* is a set of compound labels for the subject variables, i.e., a set of restrictions on the values that variables can take simultaneously. We use C_S to denote the constraints on the set of variables S .

Constraint satisfaction problem solving aims to find a single solution or all the solutions or the best solution for a given problem. The problem is to find a complete assignment of values to variables from their domains in such a way that none of the constraints are broken. In order to find a solution, constraint solvers use a systematic search, usually guided with heuristics [118]. CSP solving algorithms are simple and fast which makes them popular in a variety of problems in disparate fields, however, many problems, by nature, are constraint problems and representing them as CSPs is straightforward. Different programming languages have been developed for solving CSP problems. CLPFD (Constraint Logic Programming over Finite Domains) [21] is a library in SICStus Prolog distribution [50], developed for constraint solving. We will show in chapters 3 and 6 why and how we use CLPFD in order to tackle the problems we aim to solve.

Constraints are a set of restrictions imposed on the domain values and, in practice, can be represented as functions, inequalities, matrices, etc. Variables, either one or all, can be affected

by the constraints, and the numbers of variables being effected is called *arity* of a constraint. *Unary constraints* only effect one variable, for instance, $x_1 \neq 1$ is a constraint imposed on only x_1 and can be satisfied by eliminating 1 from the domain of x_1 . Such constraints can be discarded by a set of preprocessing steps. A constraint that effects two variables is called *Binary constraint*. All CSP problems can be written as binary constraints. They can be represented in a constraint graph with two nodes, each representing a variable and an edge joining two nodes. Another way or representing binary constraints is to use matrices, i.e., two variables with domain sizes m_1 and m_2 can be represented as $m_1 \times m_2$ matrix with 0 and 1 values for each matrix entry. For instance, at entry (i, j) , “1” indicates that both variables can simultaneously take values i and j while “0” signifies the violation of a constraint.

In case of a binary constraint $C_{i,j}$ between two variables x_i and x_j , the constraint is said to be *arc consistent* if for every value in the domain of the variables, the constraint is satisfied. The $arc(x_i, x_j)$ has a direction which is different than $arc(x_j, x_i)$. On the contrary the edge joining x_i and x_j is undirected.

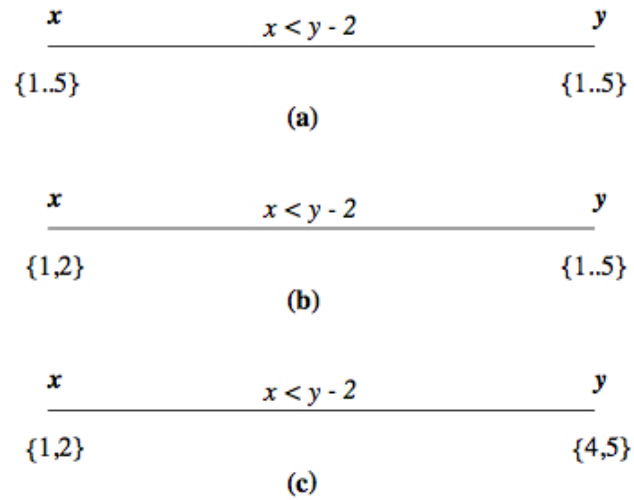


Figure 2.2: Making arc consistency in both directions (x, y) and (y, x) [118]

Figure 2.2(a) [118], reflects the original domains of x and y . It is clear than the constraint is not satisfied for all the values in the domain of x and y . For instance, if x chooses 5, there is no value in the domain of y to be at least 2 greater than x . In (b), (x, y) are arc consistent and (c) shows the arc consistency in both ways, (x, y) and (y, x) , that is, for every assignment

of x , there is a value in the domain of y that satisfies the constraint and vice-versa. Arc consistency is a reasonably cheap pre-processing step which can greatly improve the efficiency of the search algorithm by pruning the search space. Constraint solvers normally include one of many algorithms responsible for making the problem arc-consistent.

Most CSP solvers use systematic searching through values in variable domains in order to find the possible assignments. *Backtracking* is one of the most widely used techniques for searching. At each time, one variable is selected and a value is respectively assigned to it. After selecting a value, the agent makes sure that the newly selected value (label) does not contradict the former labels. The process of allocating a value is called *labeling*. In case the new label violates the constraints, then the alternative value will be selected upon availability. The problem is solved when all the variables are labeled. If at any stage none of the values can satisfy the constraints, then the formerly selected value is revised and the alternative available value will be selected. This process continues until the solution is found or all the value combinations have been tried and failed to satisfy the constraints.

One of the drawbacks of classical CSPs is when the new problem is presented as an extension to the old one with few changes in constraints and domain values. In this category of problems, where problems are *dynamic* in nature, solvers normally cannot adapt gracefully to such changes and start solving the problem from scratch. As such, all the work that has been done for solving the previous version is wasted [82].

What we described so far has been focused on *hard constraints*, i.e., the constraints are *inflexible* and all are either satisfied or violated. A perfect (valid) solution is the one that satisfies all the constraints. However, many real-life problems are over-constrained and may often result in no perfect solution. In such scenarios, the restrictions need to be *relaxed* so that at least a solution close to the expected one can be achieved. Constraints are hence prioritized and differentiated as “required” or “preferential”. This can be done by attaching costs or weights to constraints thereby dividing them to a set of *hard* and *soft* constraints. The problems we aim to tackle in our project, to some extent, can be categorized as dynamic and flexible constraints satisfaction problems (DFCSP) [81], in that, the problems are dynamic (background information keeps

changing), and in certain scenarios, there is not enough information (constraints, facts, etc) to solve the problem. However, as we will explain in chapter 3, there are other aspects to our problem types that distinguish them from being merely a constraint solving challenge.

2.5.2 Expressing Problems as CSPs

In the standard AI paradigm, as described in §2.2, a problem is expressed in a formal logic language and a particular reasoning approach is employed to solve it. The type of reasoning employed to solve a problem normally depends on the way the problem is posed: whether the problem is considered as a machine learning problem (inductive reasoning), automated theorem proving (deductive reasoning) and so on. However, there is a little fluidity in this paradigm in that a problem is presented and the solving mechanism is stuck with a chosen technique. It might be possible to model more sophisticated problem solving methodology by combining various techniques. A number of researches have so far demonstrated the benefits of combined reasoning, i.e., a reasoning process which employs various problem solving techniques for solving a given task [26, 107]. In order to do this, it is imperative to have the ability of translating a problem into the syntax of various AI systems employed.

Constraint logic programming has evolved as a successful programming paradigm in tackling a wide range of disparate fields. Therefore, problems that are not normally expressed as CSPs may also benefit from constraint solvers in cases where they can be expressed as a CSP. Charnely et. al's system, described in [24, 23, 25] and fully in [26], is composed of a translation suite whereby a problem expressed in first-order logic is interpreted as *implied* constraints in the syntax of the SICStus CLPFD constraint solver. Charnely argues that it would be possible to have a better comparison between various AI problem solving techniques by having the ability to translate a problem into the syntax of different AI problem solving systems. In addition, by expressing the problem in different domains of AI, it may be possible to employ inductive, deductive and constraint solving techniques in concert. We used this system as a part of our project described in §6.2.4 in order to translate a problem from OTTER syntax to CLPFD. The complete algorithm is explained in [26], and for brevity we only explain how the expressions

are translated into CLPFD syntax.

During the translation process each string is tokenized and parsed using the Definite Clause Grammar (DCG). The expressions are partially translated as: $e(operator, sub_exp_a, sub_exp_b)$. For instance, $(a * c)$ is translated as: $e(*, var(a), var(c))$. When an expression contains two operations, for instance, $((a * c) = b)$, the result will be a nested expression: $e(=, e(*, var(a), var(c)), var(b))$. Once parsed and partially translated, the system will be dealing with existential/universal quantifiers and solution variables. Universal quantifiers are partially translated as: $all(variable_list, sub_expression)$ which states that $sub_expression$ should hold for all the value combinations in the variable list. As an example, the translation of $all\ a\ b\ (a = b)$ will be as follows:

```
all ([var(a),var(b)], e(=,var(a),var(b)))
```

The above expression will be further translated into CLPFD syntax. This is achieved through nested expression whereby, in most instances, creates a new predicate representing each level of nesting. For instance, in translating the above expression, three predicates are constructed. The first one will find all the value combinations (possible pairs of $var(a)$ and $var(b)$) where in general can be pairs, triples, etc. The second predicates takes each combination and makes a call to the third predicate. The $sub_expression$ is represented by the third predicate. Figure 2.3 on page 22 depicts the translation of above expression in the form of constraint logic program.

D is the domain size and S is the list of value combinations for $var(a)$ and $var(b)$. The first

```
P1(D):-
    find_comb(D,2,S),
    P2(S).
P2([]).
P2([[Var1,Var2]|Ss]):-
    P3(Var1,Var2),
    P2(Ss).
P3(Var1,Var2):-
    Var1 #= Var2.
```

Figure 2.3: Example of constraint interpretation

predicate, P1, passes the list of combinations created by the pre-defined `find_comb/3` predicate to p2. Predicate p2 recursively calls p3 which constrains that all the combinations should be satisfied by the $sub_expression$. Similarly, existential quantifiers are partially translated, stating that the $sub_expression$ should hold for at least one of the value combinations in the variable list. A more detailed example of the translation and solution will be shown in chapter 6 (§6.2.4) and appendix B.1.3.

2.6 The HR Discovery System

The HR discovery system [29], named after mathematicians Godfrey Harold Hardy (1877-1947) and Srinivasa Ramanujan (1887-1920), is a combined reasoning system which performs Automated Theory Formation (ATF) and plays an important role in our work. Automated theory formation system was first introduced as a hybrid AI technique for discovery tasks [32] and has been developed and evolved, ever since, through a series of implementations resulting in HR1, HR2 and HR3 [36]. ATF title was chosen to indicate that every necessary aspect of a mathematical theory, including concepts, examples, conjectures, proofs and theorems, can be produced by this approach. HR series have been successful in different paradigms of Artificial Intelligence research such as problem solving and artefact generation. They have been used in mathematics, biology, generating music and art, etc. The details behind ATF and HR1 software which performs ATF with applications to mathematical discovery can be found in [29]. In [30], HR2 empirically finds non-existence, equivalence and implication conjectures. In this sense, the system is best described as a descriptive machine learning system. HR2 can also be projected as an Inductive Logic Programming system by noticing patterns in the data, or can have applications such as reformulating constraint satisfaction problems [25]. HR has been applied on visual, analogical representation of mathematical concepts to develop further concepts and conjectures [97]. HR has also been incorporated in other systems such as HRL [98, 95] which is a multi-agent dialogue system, extending HR by modeling the ideas of philosopher Imre Lakatos (1922–1974) [99].

HR system starts with background knowledge including concepts for a particular domain of interest and examples in Prolog notation, though it can be given background information in various different formats. It then tries to form theories using a set of *production rules* such as compose and negate to form new concepts. HR employs production rules to form new concepts from the old ones and uses conjecture making techniques to find relationships among concepts. A set of *interestingness measures* guide HR with the new concepts that are to be used with production rules [34], similar to mode declarations used in ILP systems. Other external reasoning tools can also be harnessed by HR. For instance, OTTER theorem prover [79] can be

employed to prove/disprove the conjectures produced by HR or by appealing to MACE [80] model generator, HR can find counter-examples.

2.6.1 Concept Formation

The background information about a domain of investigation can be supplied to HR as *concepts* which are the basic building blocks of a theory along with some *objects of interest*. For instance, in number theory, multiplication or addition are concepts and some integers can be objects of interest. Concepts are provided to HR with their definition and a *data-table* which contains the examples of objects of interest. Figure 2.4 is an example of the concept “love” in Cluedo game [2] which will be explained in more detail in chapters 4 and 6. As shown, the concept definition is expressed in an extended version of first-order logic. “`concept(cluedo16,[A, B, C])`” states that the HR stores the data-table of concept `cluedo16` in three columns: the first column is the `case_study` and the second and third are suspects with the relation “love”. Therefore, the concept `cluedo16` encompasses predicates whereby a truth about a single object of interest (suspect in this case) or a relation between various objects of interest (love) is expressed. A concept can also be linked to the examples of the object of interest that dis-satisfy the concept definition. For instance, figure 2.5 represents the data-table of the concept: `concept(s101_0,[A]) :- suspect(A), pred3(A), pred9(A), pred10(A)`. When an infinite number of background examples exists, HR takes a subset of the examples, assuming that the background information related to the subset is complete.

```
concept(cluedo16,[A, B, C]):- case_study(A), suspect(B), suspect(C), loves(A,B,C).

f(case-1)=[[green, scarlett], [mustard, scarlett], [white, scarlett]]
f(case-2)=[[green, scarlett], [peacock, scarlett], [white, scarlett]]
f(case-3)=[[green, scarlett], [mustard, scarlett], [white, scarlett]]
f(case-4)=[[mustard, scarlett], [peacock, scarlett], [white, scarlett]]
f(case-5)=[[peacock, scarlett], [plum, scarlett],[white, scarlett]]
```

Figure 2.4: Example of a concept definition and the examples satisfying the concept

positives	negatives
s12	s10
s21	s11
s31	s28
s40	s29
	s37
	s41
	⋮

Figure 2.5: HR data-table for the concept of murderer

2.6.2 Inventing Concepts

HR uses a set of pre-defined *productions rules* (PR) to produce new concepts by modifying or combining the existing ones. Some of the production rules can take two existing concepts as input (binary) while others take one concept as input (unary). HR is able to derive a new definition from the definition of the *parent* concept(s). For instance, the concept of prime number can be developed by HR given the background concepts of (a) an integer and (b) a divisor. HR, initially invents the concept of number of divisors for a particular integer and then inventing the prime concept by specifying the number of divisors as two. There are various production rules, each performing a unique manipulation of the existing concept(s). We only cover few of the PRs that we used in our experimentations. A more detailed list and discussion on the available production rules can be found in [29, 33, 31, 20].

- **Compose** creates a new concept by either modifying a parent concept or combining two parent concepts. For instance, by combining multiplication concept: $a * c = b$ and addition concept: $d + e = f$, HR derives a six column data-table, representing the concept $a * c = b \wedge d + e = f$. HR can also unify variables and produce more interesting concepts. For example, $a * c = b \wedge a + c = b$, is represented by a three column data-table. The choice of unifying variables is called *parameterization*. By applying compose and parameterization on multiplication, HR derives the following concept (commutativity on multiplication): $a * c = b \wedge c * a = b$.
- **Match** performs on a single parent concept and finds rows in its data-table where the columns are equals. For examples, HR can produce the concept of square numbers by

unifying a and b in $a * b = c$, resulting in $a * a = c$. This way, the data-table of the new concept will have two columns contrary to its three column parent concept. From $a * b = c$, HR can also develop $a * a = a$.

- **Negate** finds the complement of a concept. As such, it creates the negation of a concept by constructing rows which do not appear in the data-table of a concept. For instance, given the concept of square numbers, HR identifies the integers in the concept and then selects the examples in the integers data-table that do not satisfy the concept of square numbers.
- **Exists** creates a new concept by introducing an existential quantifier to a parent concept. This way, the data-table of the new concept will have fewer columns where the missing columns are quantified. For instance, by removing a column in the multiplication concept $a = b * c$, HR can derive a new concept as: $\exists c a = b * c$ with the number of its data-table columns reduced to two.
- **Forall** implements the idea in which a certain property is satisfied in all the cases. It is binary concept which modifies a combination of two existing concepts supplied by the user or takes a concept produced by HR as an input with a user-supplied concept.

2.6.3 Making Conjectures

The ideas that are in the form of propositions, statements or theories can be formulated in *assertions* which may be true or false and their truth can be reduced by derivations to that of primitive propositions. In scientific discovery, the assertions were pioneered by Karl Popper (1902-1994) to be called *conjectures* [102]. Conjectures may indicate slightly different meanings in various contexts, for instance, they may be propositions that are unproven and considered to be true or at least provisionally adopted as true. However, in this thesis, we consider conjectures as statements about various concepts which are hypothesized to be true and their truth will be verified. A conjecture is a *theorem* if its proved to be true, otherwise a *non-theorem*; it remains an *open* conjecture if its truth is undecided. One of the main research interests in the field

of Artificial Intelligence has been to devise algorithms that can make and prove conjectures [117], however, unlike automatic theorem proving, not much work has been done in automatic discovery of conjectures.

HR, tries to identify conjectures by comparing the data-tables of different concepts and empirically finding patterns between examples of concepts. For instance, if the data-table of two concepts are equal, HR makes an *equivalence* conjecture, i.e., that the definition of two concepts are logically equivalent. For instance, in the Cluedo game mentioned in §2.6.1, the concepts **love** and the negation of **dislike** have the same data-table, hence: $love(a, b) \leftrightarrow \neg dislike(a, b)$. Equivalence conjectures can also help HR prevent duplicate effort. As mentioned above, production rules modify and combine the data-table of concepts. Therefore, two concepts with an identical data-table would be developed by different production rules in identical ways. HR eliminates the repeated search by developing one of the ways. Equivalence conjectures can be used in re-writing the definition of the conjectures that are needed to be proven. Re-writing can also help HR discard and filter future equivalence conjectures.

Further, HR makes *implication* conjectures between two concepts by identifying a concept which is the specialization of another. As such, HR finds out if the data-table entries of one concept is a subset of another and suggests an implication conjecture. For instance, in the mathematics, all cyclic groups are Abelian [61] or in the Cluedo game, the data might be presented in such a way that if a hates b , a is also angry with b . In this case HR will suggest: $hates(a, b) \rightarrow angry(a, b)$. HR also makes a *non-existence* conjecture when there is no example satisfying the definition of a particular concept. In other words, when the data-table of a concept is empty, HR generates a non-existence conjecture.

2.6.4 Setting up HR

The user provides HR with background information and a set of theory formation instructions to follow. Background is stored in a *domain file* and contains the initial concepts (definitions and examples) and also, in some domains, a set of axioms to be used in conjecture proving. HR

also allows the user to write code for a specific function so that the background information can be fed to HR without manually writing it.

Every time a production rule is manipulating existing concepts with a specific parameterization, HR is taking a theory formation *step*. As a result, a concept may be generated and HR subsequently attempts to identify conjectures by comparing the new concept data-table with the existing ones. HR can invoke OTTER to prove conjectures and in case OTTER fails, HR can harness MACE [80] to find a counter-example. If neither of them are successful, the conjecture remains open. HR also stores the proved conjectures to be used later in order to prove theorems without the use of OTTER. We should bear in mind that the newly generated concept may have the same definition as the existing concepts, in that case, HR discards the concept.

The role of user is to determine a set of parameters for HR prior to the start of a theory formation session to control HR while constructing a theory. The controlling mechanism contains a set of instructions describing how and which of the production rules should be used in identifying the conjectures. In general, a set of theory formation instructions are supplied to HR as controlling *agenda*. These instructions contain a hierarchical list of steps, each step indicating the next instruction that should be executed. The agenda also instructs how HR should assess the concepts.

HR can exhaustively search for the new concepts by using all the concepts and production rules and exhausting all possible parameterizations. However, it can use a list of measures to evaluate the interestingness of a concept. Interestingness measures include *parsimony* of a concept which is inversely proportional to the size of the data-table for each concept, *complexity* which is proportional to the number of production steps used in building a concept, *novelty* which is inversely proportional to the number of other concepts that generate the same *categorization* and so on. A category contains HR's set of groups that are considered equal according to the description of the concept's data-table. Concepts that introduce new categories are interesting.

Although HR acts autonomously during the theory formation, the user can halt the session by direct intervention. The user can also ask HR to build a theory for a certain number of things, for instance, a 100 conjectures, 200 concepts. The user can also force HR to finish the task after

a certain number of steps, or to build a theory for a certain period of time.

2.6.5 HR3

So far, we briefly discussed the concept behind HR and the mechanism it employs in its earlier versions, HR1 and HR2. HR, in general, employs production rules to turn old concepts into new ones and empirical conjecture making techniques to find relationships between the concepts. Largely applied to mathematical invention tasks, but also with applications in other domains, HR has been used with much success. However, in chapter 9, we will show some of the HR2 limitations with regard to this research. In particular, the speed at which HR2 operates and its memory consumption have held it back in domains with large search space, such as the one we investigate in our project. Given these failures, HR3 [36] has been built from scratch in order to overcome the drawbacks of its previous versions while the memory footprint and search efficiency have been the paramount considerations in the new design. The problems we will introduce in this thesis have made some of HR2 deficiencies apparent and have been one of the driving forces behind the development of HR3. These problems have also provided a benchmark for comparing HR2 and HR3. The results of the comparison will be illustrated in chapter 9. Few of the design considerations are briefly highlighted below.

One of the main features that has enabled HR2 to be presented as a creative system is the ability to generate various types of output [37, 96]. HR3 has therefore maintained and improved this characteristic in its design. As such, more production rules have been implemented in HR3 in order to deal with textual data such as tweets or to handle floating point numbers. Additional production rules also allow HR3 to read input in more diverse formats including Prolog and CSV files, Java code and ontologies. Another design consideration in HR3 was to maintain and improve the lack of reliance on logical formalisms HR2 benefits from; HR2 uses two separate processes while (a) generating data for a concept and (b) generating definition(s) for a concept. This has enabled HR2 to be applied to the tasks where restrictions would be imposed by a logical representation.

However, the main feature that has distinguished HR3's performance, with respect to the

problems we tackle in this thesis, is the ability to perform *on-demand* [36]. Considering its predecessor, the majority of HR2's processing is done to no avail. For instance, HR2 consumes a lot of the resources in order to form concept definitions that no-one ever reads, or conjectures that no-one will investigate. It also employs third party systems to prove/disprove conjectures that no-one is interested. In all the mentioned tasks, HR2 spends an inordinate amount of time and consumes excessive memory. HR3, on the contrary, is designed in such a way that it performs the barest amount of work to form a theory, and the extra processing is performed only after the user chooses which of the material they are interested to look at. For example, HR3 does not produce the definitions during the theory formation session and while it records non-existence and equivalence conjectures, it does not formulate them either. In contrast to HR2 that pushes the memory boundaries while dealing with a large background theory, HR3 is designed to keep the memory footprint as low as possible. Overall, implementing the mentioned differences has made HR3 much faster at building theories with more value compared to HR2. In chapter 9, we will further investigate HR3's performance comparing to HR2. For a fuller discussion on the case studies leading to HR3 development, and more detailed description on design considerations and implementations of HR3, see [36].

2.7 Machine Learning

In cognitive science and related fields, the term *learning* is used to express the process in which the information is obtained through observation [115, 8]. Learning in humans is mostly composed of memorizing, undertaking comprehension tasks and learning from examples. A branch of Artificial Intelligence, namely Machine Learning, is dedicated to devise algorithms that allow automated learning. According to Tom Mitchell, the author of a standard machine learning text [83], "Machine Learning is the study of computer algorithms that improve automatically through experience". Machine learning is applied in a wide range of application domains including: medicine, bioinformatics, search engines, handwriting recognition, natural language processing, military, arts, etc. In this section we cover some of the background concepts in the field of machine learning which are essential for understanding this research. A detailed

introduction of machine learning can be found in [83].

A machine learning agent is typically used to automatically extract knowledge and experience from the environment by learning from the examples in the form of *inductive reasoning*. Inductive reasoning is a method in which the inferences are made with less confidence compared to deductive reasoning. In deductive reasoning, as explained in §2.2.2, the conclusion is entailed from the premises in case of a valid deduction, i.e., the truth of the premises guarantees the truth of the conclusion. In inductive reasoning, however, the accuracy of the premises guarantees the correctness of the conclusion with some “*degree of strength*” [57].

Given some background knowledge (a set of examples and some information about the examples), machine learning aims to use inductive reasoning to infer new relevant information. Two main areas of machine learning are: (a) unsupervised learning and (b) supervised learning. In unsupervised learning, given a set of unlabeled examples, a machine learning agent is harnessed to determine how the data should be organized. Unlabeled examples are instances with no “explanation” whereas labeled data, consists of unlabeled examples that are augmented by some sort of information such as tag, class or label. The common task in unsupervised learning is *clustering* which is to allocate the given data to a set of distinct clusters. Each cluster contains data instances that are more similar within that cluster than to instances in other clusters.

In supervised learning, a set of labeled data with the desired target output is provided to the system. The aim of a supervised machine learning agent is to learn a function (*hypothesis*) that generalizes from the supplied examples to the unseen data. As such, the function learns why the examples are put into certain categories (labeled with certain tags) and tries to predict the category in which unseen examples should be allocated. For instance, given a labeled sample of animals, each containing attributes (e.g. *flies*, *has_four_legs*, *has_feathers*, *barks*), the system may be to learn the concept of “bird”.

The examples in supervised learning are typically separated into *training* and *test* sets. Training set consists of labeled instances that are used while learning the hypotheses. The test set, on the other hand, is never seen during the learning process and is used to test the accuracy of the yielded hypotheses in predicting the categorization of the unseen examples. Of course, there

are some other machine learning techniques, such as *n-fold cross validation* in which the entire set of examples is used for learning and testing purposes. As such, examples are randomly partitioned into n equal sizes, each partition is then hold back for testing and $n - 1$ partitions are used for learning. The overall predictive accuracy is averaged over all percentages produced at each testing stage [83].

For a better understanding of the supervised learning, let us consider the following mathematical notation: (1) E is the set of examples where each instance is represented by e , (2) c is the *target concept* that is presented to the system and is aimed to be learned and (3) the target concept value of each instance e is $c(e)$. Machine learning system is to find a hypothesis, h , such that $h(e) = c(e)$. Therefore, the system searches through a set of possible hypotheses in order to the find the hypothesis that best fits the provided examples and background knowledge/constraints. Indeed, a machine learning is assessed by its predictive accuracy over the unseen examples. In machine learning, the central assumption is that any hypothesis that best describes the target concept over a sufficiently large training set, will also perform well for the unseen examples, unless it memorizes the data labels (*over-fitting*) rather than generalizing the concepts.

We discussed automated deduction and resolution in §2.2 and §2.4. As resolution is complete, the researchers attempted to invert it in order to construct the inverse entailment operator. Inverse resolution is an inductive reasoning technique, employed by many ILP systems [83] (we will explain ILP systems in the following sections). Let us consider the propositional resolution operator once more for simplicity. Given two arbitrary clauses $C_1 = A \vee B \vee D$ and $C_2 = E \vee \neg B \vee F$, the resolution operator initially finds a literal which appears as negative in one of the clauses and positive in another (common literal). In order for the conjunction of C_1 and C_2 to be true, if B is false then $A \vee D$ should be true and if B is true, then $E \vee F$ should be true since $\neg B$ is false. Therefore it concludes that $A \vee D \vee E \vee F$ is true, i.e., $A \vee D \vee E \vee F$ is the resolvent. Now, given the resolvent C and an initial clause C_1 , the inverse resolution operator finds a clause C_2 such that $C_1 \wedge C_2 \vdash C$. By the definition of resolution, whatever literal appears in C and not in C_1 , it must have been present in C_2 , and any literal L that is present in C_1 and not in the resolvent C must have been present in the C_2 in its negated form. The difference of this process in first-order logic is the addition of the notion of *unifying substitution*.

A substitution is any mapping of variables to terms and $L_1\theta = \neg L_2\theta$ for two literals L_1 and L_2 , signifies θ as a unifying substitution. Note that θ can be factorized to two different sets of substitutions (substitutions on C_1 and C_2 , i.e., $\theta = \theta_1\theta_2$). Summing up, inverse resolution operator, given the resolvent C and the initial clause C_1 , finds C_2 as follows:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\}$$

Figure 2.6: Inverse resolution rule in first-order logic

2.7.1 Data mining and Association Rule Mining

Knowledge Discovery in Databases or simply *Data Mining* is considered as a subfield of computer science which draws on ideas from machine learning, artificial intelligence, statistics and data base processing. Although it may often be considered as a distinguished field by itself, in most cases follows the same principles as in machine learning. The slight differences with machine learning lie in the perspective or the background of the problems. In this project, we have the same view and expectations on both data mining and machine learning techniques, however, each of these approaches are employed differently depending on how the problems can be modeled/represented. The way the problems are represented for machine learning and data mining systems, typically follows a hierarchy by which their degree of expressiveness is indicated. The hierarchy normally starts with boolean representations (BL) with the least expressiveness power, and ends with logic programs (LP) as the most expressive representation scheme. Attribute-value representations (AV), multi-instance (MI) and relational representations (RR) fall in between.

In machine learning, as described earlier, the task is to learn the function that generalizes from the supplied examples to the unseen ones. In its simplest form which is binary classification, the machine learning agent is to learn how to discriminate the unseen examples by allocating them to positives or negatives. A *loss function* measures the quality of the learned function (hypothesis) in order to minimize the error. As such, the loss function may be to calculate

the number of instances that have been classified wrongly in the training set, or to calculate the least mean square. In data mining there is normally a slight difference: the aim is to learn the hypothesis that satisfies a given quality criterion, for instance, the criterion would be for the hypothesis to cover 90% of the data. Often, the hypothesis should satisfy a wide range of quality criteria. Data mining is to find a single element, n element, or all elements in a dataset that satisfy one or more quality criteria. Instead of loss function in machine learning, here in data mining, a commonly used function is the *frequency function*. For instance, the number of times that the hypothesis is satisfied by the dataset. An important problem in data mining paradigm is to find interesting patterns or implicit regularities in datasets, i.e., *association rule discovery* [42, 132].

Identifying the frequent itemsets in a database and establishing rules (conditional implications) amongst them was first introduced as *Association Rule Mining* in [6]. Given a number of objects (database), association rule mining is to find a set of attributes shared amongst the objects. For instance, consider the hospital episodes as database, where the objects are the patients and each patient's medical history (symptoms, test results, operations and diseases) is a set of attributes. Association rule mining can be used to find the common attributes amongst patients. A couple of rule examples could be that: 70 percent of the patients who ate fish in restaurant X within the last couple of days, have food poisoning, or 90 percent of the patients who had colorectal cancer surgery, suffered from some sort of complications two weeks after the surgery. These type of rules can be used in numerous ways: to evaluate the performance of a hospitals, to measure the usefulness of a treatment, etc. Association rule mining can also be used to learn the classification rules. One of the first applications of association rule mining was to help deciding on marketing strategies, for instance, finding the common items that are bought together convinced the supermarkets to stock them in a close proximity, thereby increasing their sales. However, association rule mining is currently a well researched technique, commonly used in a wide range of domains including bioinformatics, web usage mining, etc.

In the following section, we elaborate on one of the common algorithms of association rule mining that is mostly used for problems with attribute-value representations. This type of representation is more suitable when we need to explain the characteristics of examples. For

instance, describing daily transactions of a supermarket. Further, in §2.8, we will describe logical and relational learning in more details and also ILPs that benefit from a more expressive framework. This type of representation is more appropriate when we need to learn a specific concept.

Although, in the future chapters, we will discuss the nature of the problems we aim to solve, it is worth mentioning that those problems can be more naturally expressed in first-order logic. Nonetheless, we also investigated the possibility of modeling our problems with attribute-value representation in order to investigate the possible outcomes of using of simpler representation frameworks on our datasets. You can find the full discussion, results and the detailed analysis in chapter 9 (§9.3).

2.7.2 Apriori Algorithm

Apriori is considered to be the most widely used algorithm for association rule mining since its first introduction in [6]. Before discussing Apriori algorithm, it is needed to initially describe the formal definition of some of the basic terms. Let $I = \{I_1, \dots, I_m\}$ be an itemset which is a collection of one or more distinct literals called items. Note that k -itemset indicates an itemset with k items. The database contains a set of transactions $T = \{t_1, \dots, t_n\}$ where each t_i consists of an array of size m . for instance, $t_i[k] = 1$ states that the item I_k appears in the transaction t_i , otherwise $t_i[k] = 0$. A transaction t_i *satisfies* an itemset X , if for all the items I_k appearing in X , $t_i[k] = 1$. The task of association rule mining is to extract the implications from the database in the form of $X \rightarrow Y$, where X is an itemset and Y is a single item which does not appear in X , i.e, X and Y are not intersecting. The initial approach to association rule mining, known as *brute-force*, is to take two steps for extracting the interesting rules: (a) imposing *support* constraint known as frequent itemset generation and (b) imposing the *confidence* constraint and pruning the rules that break it, known as rule generation. These two factors evaluate the interestingness of the rules.

Support constraint guarantees that the fraction of transactions that satisfy the items in both X and Y ($X \cup Y$), is more than a user-defined threshold (*minsupport*). Confidence constraint, on

the other hand, highlights the implications that satisfy the user-defined threshold of *confidence factor* known as *minconfidence*. The rule $X \rightarrow Y$ has a confidence factor $0 \leq c \leq 1$, if and only if at least c % of the transactions that satisfy X also satisfy Y . The confidence factor can be stated similar to a conditional probability $P(Y|X)$ and is calculated according to the following formula:

$$Conf(X \rightarrow Y) = \frac{supp(X \rightarrow Y)}{supp(X)} \quad (2.1)$$

This approach is computationally expensive as the total number of candidate items, given m unique items, would be $M = 2^m$. That is, the complete search would require exploring 2^m nodes. The complexity of this algorithm is $O(NMw)$ where N is the size of transactions in T and w is the maximum number of items in a transaction. Given m items, the total number of association rules is calculated according to the formula 2.2 [132]. For instance, when $m = 6$, the total generated rules is 602.

$$\text{Total Rules} = \sum_{k=1}^{m-1} \left[\binom{m}{k} \times \sum_{j=1}^{m-k} \binom{m-k}{j} \right] \quad (2.2)$$

The Apriori algorithm tries to resolve the high complexity issue by taking a few extra steps. First and the foremost, the Apriori principle is based on *anti-monotone* property of support. This property is stated in definition 2.1.

Definition 2.1. anti-monotone property: Let X and Y be two itemsets and let s represent the support. $\forall X, Y : (X \subseteq Y) \rightarrow s(X) \geq s(Y)$. The formulae indicates that the support of itemset is never greater than the support of its subset.

With the help of anti-monotone property of support, Apriori algorithm can efficiently prune the search space in its breadth-first search approach; if an itemset is infrequent ($support < minsupport$), all its supersets are infrequent and should be discarded, therefore, $M = 2^m$ is reduced.

Apriori uses $(k - 1)$ -itemsets to generate k -itemsets. For example, 1-itemsets are considered for generating 2-itemsets. This way Apriori avoids generating repeated itemsets. After generating

each candidate itemset, Apriori stores them in a *hash-tree* structure [64]. As such, Apriori avoids scanning through the whole database for calculating the support of the generated itemsets and only compares them with the candidates in the hash-tree. Therefore, the number of comparisons (NM) in the complexity $O(NMw)$ is reduced due to the fact that N is reduced with the help of hash-tree. The details of Apriori algorithm can found in [6, 7] and [60] provides a brief survey of various implementations. For another popular association rule mining technique based on lattice traversal techniques, see [56].

In addition to large search space, support-confidence framework has certain drawbacks. In case the threshold for support (*minsupport*) is set too high, interesting but rare itemsets may get discarded. On the contrary, if the *minsupport* is set too low, the large number of itemsets will be computationally expensive to search and will result in too many patterns. Low minimum support may also spawn spurious high confidence patterns. The main drawback of confidence is that the support of the consequent is ignored. As such, the negative correlation between the consequent and antecedent may get concealed by a high confidence, and the rule will be considered as interesting. This can be resolved with the use of *lift* which is calculated according to the following formula [19]:

$$Lift(X \rightarrow Y) = \frac{conf(X \rightarrow Y)}{supp(Y)} = \frac{conf(Y \rightarrow X)}{supp(X)} = \frac{P(X \cup Y)}{(P(X)P(Y))} \quad (2.3)$$

In case of $lift > 1$, there is a positive correlation between antecedent and consequent; $lift = 1$ states that both X and Y are independent and $lift < 1$ implies a negative correlation. Another interestingness measure is *conviction* that was first introduced in [18] as follows:

$$Conviction(X \rightarrow Y) = \frac{1 - supp(Y)}{1 - conf(X \rightarrow Y)} = \frac{P(X)P(\bar{Y})}{P(X \cup \bar{Y})} \quad (2.4)$$

Conviction is somehow similar to lift in that it compares the probability of X without Y with their frequency. However, unlike lift, it is a directed measure. Leverage introduced in [101] reveals the difference between the probability of X and Y appearing together and their independent probability. In application domains such as marketing, it is useful to know how much more X and Y have been sold together comparing to their individual sales. Leverage is

determined as follows:

$$\text{leverage}(X \rightarrow Y) = P(X \cup Y) - (P(X)P(Y)) \quad (2.5)$$

There are various criteria for evaluating the interestingness of a rule. In general, a useful criteria is known to have one or both of the following properties: (a) generality and (b) reliability [53, 125]. Support or coverage represents the generality and confidence can depict the reliability of a rule. Therefore, the combination of support-confidence framework provides a useful measure of interestingness which makes it one of the most popular standards. In our project, however, we will employ the rest of the commonly used measures introduced in this section and we will illustrate and discuss their performance in chapter 9 (§9.2.7).

2.7.3 Weka

Waikato Environment for Knowledge Analysis, *Weka*, is a Java based machine learning suite and is commonly used by a wide range of researchers. Weka benefits from set of data-preprocessing tools, learning algorithms and evaluation methods. The significant advantage of this program, is the ability to accept data with numerical as well as nominal attributes. It is also possible to import clustering, classification and many other algorithms which are not built-in by default. We use Weka in our project due to its built-in Apriori algorithm complemented by additional interestingness measures. Table 2.5, taken from [5, 74], contains the adjustable parameters that we will use in our project for various settings. Setting and experimentation details are shown in chapter 8.

2.8 Logical and Relational Learning

Traditional and early machine learning techniques have so far exhibited serious limitations from knowledge representation perspective. For instance, association rule mining [6] or decision trees [104] can only perform in a certain language representations. Such systems, largely employing

Parameter	Specification
$-N$	required number of rules (default =10)
$-T$	$0 = confidence, 1 = lift, 2 = leverage, 3 = conviction$
$-C$	minimum confidence (default = 0.9)
$-D$	delta by which the minimum support increases at each iteration (default = 0.05)
$-M$	lower bound for minimum support
$-U$	upper bound for minimum support
$-V$	forcing the program to report progress iteratively
$-t$	name of training set

Table 2.5: Some of Weka parameters for Apriori algorithm [5]

propositional representations, cannot elegantly express complex problems in which there are multiple entities that hold many relations amongst them. For example, expressing a chemical compound in order to predict its reactions, or representing a social network in order to find a pattern of social interactions, is not straightforward with propositional representation due to its restrictions that can only represent true or false facts. As such, a subfield of artificial intelligence was motivated and emerged which encompasses relational or first-order logic as expressive formalisms in order to tackle machine learning problems. This field, described by many as *logical and relational learning*, is built upon combining two major subfields of artificial intelligence, i.e., machine learning and knowledge representation [42]. There are various names by which the logical and relational learning is often referred, such as, multi-relational data mining, statistical learning or inductive logic programming. Even though there are slight differences in the formalisms each employs, they essentially tackle the same domain of problems using *inductive inference*. Recalling from §2.7, in inductive inference, unlike deduction, the conclusions, though cogent, do not logically entail from the premises. In the following section, we briefly discuss the inductive logic programming.

2.8.1 Inductive Logic Programming

Inductive reasoning, by which a set of specific facts are generalized into laws, forms the basic of many scientific theories and has been the subject of study by many philosophers and scientists. The idea of scientific discovery using inductive inferences was to be empirical. That is,

the knowledge should be obtained through observation, set of experiments, generalization and testing the hypotheses. As described above, the field of logical and relational learning aims to harness the computational approaches to inductive reasoning and scientific discovery. Scientific theories, however, are typically of complex nature and representing them in an expressive way across various domains was a computational difficulty. Another obstacle in supporting a scientific discovery was for the computational techniques to make use of the available background knowledge. Motivated from these two obstacles, *Inductive Logic Programming* was built as a computational approach to inductive reasoning with an expressive formalism and the ability to employ the background knowledge [42].

Various researches contributed to this sub-field of artificial intelligence amongst which the Model Inference System [116] is one of the earliest inductive logic programming systems *avant la lettre*, and still remains as a powerful tool for program synthesis and theory revision. Contributions of Stephen Muggleton [86] in encompassing the automatic programming and machine learning within the computational logic framework was a breakthrough in defining inductive logic programming (ILP). [93] provides a thorough introduction to ILP and its heavily mathematical fundamentals.

As the name inductive logic programming suggests, the problems are expressed in logic programs. Recalling from §2.8, ILP is often considered as an intersection between machine learning (to be more precise, inductive concept learning) and logic programming. Inductive logic programming, in brief, aims to *induce* (discover) the definition of a *target* concept by observing a set of positive and negative examples and the possible background information, all presented in the form of logic programs. The background information, called as *background knowledge*, appears as horn clauses or facts (horn clauses with no head), providing further information pertaining to the target concept. The instances of the target concept are supplied to the system, each with a label that indicates whether the instances satisfy the concept definition or not, i.e., positive and negative examples. Similar to background knowledge, supplying the negative examples to the system is also optional.

2.8.2 The normal ILP setting

Logic programs representing the components of an ILP problem in mathematical notations, are: (1) E^+ as a set of positive examples, (2) E^- as a set of negative examples (3) B as a set of horn clauses and facts representing the background concepts and (4) H is the logic program representing the learned hypothesis (definition of the target predicate). Note that E^+ and E^- are normally ground facts. The aim of an ILP system is to learn H such that together with B entails all the positive examples and none of the negative examples.

According to [93], the normal ILP setting is defined as: given a set of finite clauses B as background knowledge and disjoint sets of clauses representing positive and negative examples E^+ and E^- , find a theory H (set of clauses) such that $H \cup B$ is *correct* with respect to E^+ and E^- . A theory is correct if it is *complete* with respect to E^+ (covers and explains the positives) and *consistent* with respect to E^- , therefore $H \models E^+$ and $H \not\models E^-$. We should note that even though the ultimate goal of an ILP system is to induce a correct theory, but this may not be always practical due to possible noise in the examples and the background knowledge present in real-life problems. Hence, the ILP systems are normally built with more flexibility to compensate for the noise.

A problem should initially satisfy a set of *prior conditions* before the learning process starts. Satisfying these conditions, the problem is guaranteed to have a solution which is not already available by the background knowledge.

$$\begin{array}{ll} \forall e \in E^-(B \not\models e) & \text{prior satisfiability} \\ \exists e \in E^+(B \not\models e) & \text{prior necessity} \end{array}$$

The set of conditions for a *correct theory* explained above, is also called *posterior conditions*:

$$\begin{array}{ll} \forall e \in E^-(B \wedge H \not\models e) & \text{posterior satisfiability (consistent)} \\ \forall e \in E^+(B \wedge H \models e) & \text{posterior sufficiency (complete)} \end{array}$$

In inductive learning, the goal is to find a theory that generalizes from the supplied examples in order to have a good predictive power for the unseen ones. Therefore, a learned theory H where $H = E^+$ is a lengthy and trivial one which classifies all the unseen examples as negative, i.e., no predictive power. Many ILP systems therefore employ a kind of heuristic such as Occam's Razor [119] and some measurements to evaluate the merit of a learned theory, thereby avoiding the trivial ones. According to Occam's Razor principle, the *simplest* of the learned hypotheses, all having the same predictive power, is chosen to be the best hypothesis. In ILP setting, the simplicity is evaluated by *minimum description length principle*, i.e., when the *example coverage* is equal, the shorter hypothesis is preferred to the longer ones. The example coverage typically evaluates the compression of data and is calculated as the weight of the positives covered by the theory, minus the weight of negatives covered, minus the number of literals presented in the theory itself. Recalling the case $H = E^+$, H is not compressive and hence trivial as it has the same number of literals originally explaining all the positive examples. In chapter 9 (§9.2.6), we will elaborate more on Occam's Razor and will argue why in some cases including the problems we will tackle in this thesis, ILP systems should discard the Occam's Razor principle.

In order to find a correct theory, ILP systems employ an specific search direction in their search methods that can be depth-first, breadth-first or even heuristics such as hill-climbing. Depending on the search direction, ILP systems are considered to have two major approaches: top-down and bottom-up. However, there are ILP algorithms that employ a combination of both approaches as a bi-directional search [120]. In top-down approach an overly general theory is initially selected and is successively specialized. Top-down approach typically results in theories with shorter and more general clauses. In the bottom-up approach, on the contrary, an specific theory is generalized to cover more examples. This approach leads to theories with long clauses.

Definition 2.2. The hypothesis h_1 is more general than h_2 , if and only if all the examples that are covered by h_2 are also covered by h_1 . That is, if $c(h_1)$ and $c(h_2)$ be the set of examples covered by h_1 and h_2 respectively, then $c(h_2) \subseteq c(h_1)$. This can also be denoted as $h_1 \preceq h_2$.

In both top-down and bottom-up approaches, a *refinement operator* plays a crucial role whereby the theories are generalized or specialized. This operator performs certain changes, for instance,

adds a literal to the body of a hypothesis to specialize it or removes a literal to generalize.

In this section as the name suggests, we discussed the normal ILP setting in which the aim is to find a theory that entails all the positives and none of the negatives. This setting is also known as *predictive learning setting*. Another ILP setting is non-monotonic which is not in the scope of this research. This setting, often called *descriptive learning setting* is mostly applied to knowledge discovery and data mining domains. The goal is to induce theories that are true for the examples, i.e., the theories explain interesting regularities in the given examples (background concepts), while the theories may not entail all the positive examples. The descriptive ILP systems such as Claudien [43] have essentially similar goal to HR's (as discussed in §2.6), though they employ different approaches. Some of the non-monotonic settings are discussed in [44].

In the following section, we will briefly discuss an important aspect of ILP systems: search space. A partially ordered set between least upper bound and greatest lower bound of two hypotheses is called *lattice* which plays an important role in ILP system's efficiency. The search space is the same as lattice under θ -subsumption; for more details on the concept of lattice, see [41]. Refinement operators, θ -subsumption and lattice structures are fundamental and heavily formal (mathematical) concepts of ILP and are not in the scope of this document. However, we highly recommend [42] for further reading on machine learning, data mining and ILP in general.

2.8.3 Defining and Bounding the Hypothesis Search Space

To recapitulate, the ILP algorithms, in general, perform the following tasks while learning the hypotheses: (1) initialize a set of theories, (2) select the most promising theory (candidate theory), (3) apply the refinement operator to generalize or specialize, (4) prune the trivial and unwanted theories and (5) check whether the generated theory satisfies the conditions of a correct theory (complete and consistent). The tasks mentioned in (2) and (4) are the constituents of the search strategy employed by the system. In this strategy, the search space is defined by *mode declarations* which aims to bias and determine the boundaries of the search

space. There are two types of mode declarations: *head* and *body*. Head mode declaration, *modeh*, determines the head of the hypothesis the system needs to induce (target predicate), and body mode declarations, *modeb*, state the literals that may appear in the bodies of the learned hypotheses.

Mode declarations also provide further information about the types of arguments appearing the hypothesis. Each argument of the mode declaration predicates is associated with a +, − or # sign by which the type of argument is stated. The prefix “+” indicates that the argument is input and should be instantiated before the predicate is called. The prefix “−” indicates that the argument is output. The output arguments may be input in another predicates, i.e., a variable will be returned as output and may be used later in another predicate as input. Finally, the argument associated with the prefix “#”, is a constant and will appear directly in the hypothesis body. Such restrictions in the form of mode declarations significantly reduce the size of search space.

With the mode declarations, the ILP system will be informed by the *recall* parameter whether a predicate appearing in the hypothesis is determinate or non-determinate. If the recall is set to 1, the predicate is determinate. That is, given a particular instantiation of its arguments, the predicate succeeds at most once. The recall value more than one, states a non-determinate predicate. Given any instantiation of the input arguments, a non-determinate predicate may succeed more than once. The symbol “*” indicates that the recall is unbounded and the predicate associated with this symbol may be succeeded in any arbitrary number of times. However, this symbol is internally bounded with a typically large integer. The integer varies depending on the ILP system, for instance, it is 100 in Progol [110].

An important concept in ILP is the *bottom* or *starting* clause, denoted by $\perp$. The bottom clause is the *most-specific clause* of an example within the hypotheses space and is crucial in bounding the hypothesis space. In other words, ILP system constructs the bottom clause to bound the search for a clause that covers an example so that any hypothesis covering the same example will be more general than the bottom clause. The hypothesis that is not more general than $\perp$ can therefore get discarded. In a top-down approach, $\perp$ is at the bottom of the space (hence the

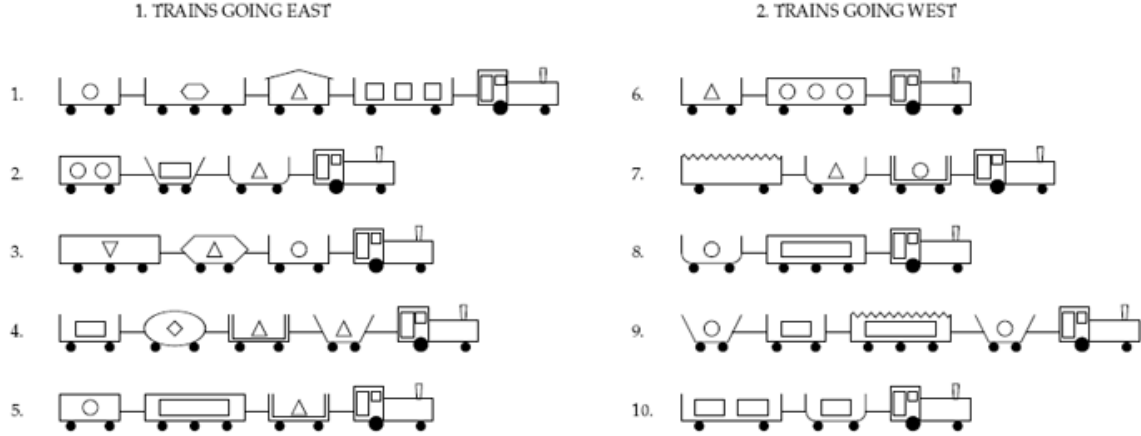


Figure 2.7: Michalski train problem

term bottom clause), whereas in the bottom-up approach, it is the first clause to be considered, hence at the top of the hypothesis search [42]. When constructing the bottom clause, a user-defined parameter, “ i ”, controls the number of iterations (layers) in which the variables are considered. For instance, if $i = 1$, the bottom clause will only contain the variables appearing in the head of the clause (layer 0). Therefore, at each “ i ”, the variables in the layer $i - 1$ are added to the most-specific clause. Choosing a small value for “ i ” will deprive the bottom clause from having all the background facts. We will discuss this in more details shortly.

Let us consider the toy example known as Michalski’s train problem [71] as shown in figure 2.7 on page 45. With the use of this example we can illustrate some of the ILP concepts discussed in this section. According to the figure, there are two sets of trains: eastbound and westbound. The purpose of the problem is to learn the target concept by which the trains can be classified as eastbound or westbound depending on their composition. The mode declaration for this problem are shown in figure 2.8.

<code>modeh(1,eastbound(+train)).</code>	<code>modeb(1,open(+carriage)).</code>
<code>modeb(*,has_carriage(+train,-carriage)).</code>	<code>modeb(1,wheels(+carriage,#int)).</code>
<code>modeb(1,closed(+carriage)).</code>	<code>modeb(1,infront(+train,-carriage)).</code>
<code>modeb(1,short(+carriage)).</code>	<code>modeb(1,infront(+carriage,-carriage)).</code>
<code>modeb(1,long(+carriage)).</code>	<code>modeb(1,load(+carriage,#shape,#int)).</code>

Figure 2.8: Michalski’s train problem: mode declarations

The background knowledge to an ILP system, as discussed earlier in this section, consists of

facts and can also contain general concepts (rules), expressed in Prolog notation. In case we need to learn the concept of eastbound trains, the labeled examples, in Prolog notation, will be as shown in figure 2.9. Bear in mind that the examples preceded by the symbol `:-/1` are negative.

<code>eastbound(east1).</code>	<code>eastbound(west6).</code>
<code>eastbound(east2).</code>	<code>eastbound(west7).</code>
<code>eastbound(east3).</code>	<code>eastbound(west8).</code>
<code>eastbound(east4).</code>	<code>eastbound(west9).</code>
<code>eastbound(east5).</code>	<code>eastbound(west10).</code>

Figure 2.9: Michalski's train problem: positive and negative examples

The following clause represents the target concept of the Michalski train problem shown in figure 2.7:

```
eastbount(X):- has carriage(X,Y), closed(Y), short(Y).
```

Following our discussion on “*i*” parameter, it is clear from the clause that the variable *Y*, first appears in the second layer. Therefore, given $i = 1$, the predicates `closed/1` and `short/1` will not be presented in the target concept as they will not be included in the bottom clause. The appropriate setting for the ILP system to learn the target concept, in this case, is $i = 2$. However, we should note that the increase in “*i*” exponentially increases the search space. While discussing the results of our project in chapter 9, we will elaborate on the difficulties that become apparent with the increase of search space in ILPs (§9.3).

2.8.4 Inverse Entailment and Progol

Although the inverse entailment we discussed in §2.7 is a fascinating method in generating the hypotheses in inductive reasoning, but in practice it can yield a large set of hypothesis during the learning process. Progol, as a land mark ILP system introduced in 1995 [84], is one of the first systems to tackle this problem. It has inspired many ILP systems ever since and various systems have been incorporating and adopting its ideas ever since. Progol employs Mode-Directed Inverse Entailment (MDIE) algorithm to generate the bottom-clause (most specific hypothesis) that along with the background information, entails the examples. Throughout the hypothesis

search space, bottom-clause guarantees a general-to-specific search bound. Therefore, only the hypotheses that are more general than the search bound will be considered. That is, the literals appearing in the body of the learned hypotheses, are the generalizations of the bottom-clause. Input to Prolog consists of background knowledge B , a set of metalogical statements, M , characterizing the form of the hypotheses and a set of examples E , all as logic programs denoted by the tuple $\langle M, B, E \rangle$. The goal of Prolog is to learn a set of hypothesis H such that each $h \in H$, given the background knowledge, covers at least one positive example $e \in E$, i.e., $B, h \models e$ and none of the negative examples. This equation can be rearranged by the law of contraposition as $B, \neg e \models \neg h$. The new equation is applicable to Prolog, as such, the hypotheses are derived from the background and examples using proof by refutation. Selecting an example e , Prolog construct its related bottom clause $\perp_e$. Having explained the bottom clause, the following will hold: $B, \neg e \models \neg \perp_e \models \neg h$. Prolog will then search for the hypothesis, employing a general to specific search through the hypothesis space bounded by the most specific clause $\perp_e$ and the most general clause (empty clause): $\Box \prec h \preceq \perp_e$ [110, 90].

Within the set of hypotheses, Prolog employs “Occam compression” [84] to look for the simplest (having shortest length) hypothesis with the help of an admissible heuristic A^* algorithm. Employing the A^* , Prolog prunes away the trivial hypotheses while guaranteeing that the shortest ones are not discarded. Prolog’s search lattice may become large depending on the $\perp_e$, as such, only a small segment of the lattice will be searched.

We compared the concept of more general/specific clauses in definition 2.2. Following that, it is worth mentioning that there are various frameworks for generality. Considering two clauses A and B , we say that A is more general than B if and only if $A \models B$. In other words, B follows deductively from A and A follows inductively from B (inverting the deductive operator). The frameworks depend on the form of A and B , whether they are single clauses, first-order theories, etc. The most popular and important framework for generalization and specialization, used by almost all the ILP systems, is θ -subsumption, and is defined as follows [93]:

Definition 2.3. θ -subsumption Let C_1 and C_2 be definite clauses with C'_1 and C'_2 as their corresponding sets of literals. We say that C_1 θ -subsumes C_2 , denoted by $C_1 \succeq C_2$, if and only

if there exists a substitution θ such that: $C'_1\theta \subseteq C'_2$. C_2 is more general than C_1 according to definition 2.2

According to definition 2.3, in case $C_1 \succeq C_2$, all the literals appearing in θ substitution on C_1 should also appear in C_2 . The problem of finding θ -subsumption given C_1 and C_2 is hence, to find a substitution θ by which the set of literals of C_1 is a subset of C_2 literals. We should note that even though in machine learning, the longer clauses are typically more specific, this is not always true when dealing with θ -subsumption [42].

2.9 Stochastic Logic Programming

The structure of data in many real-world learning and data mining problems is complex and the ability to reason about uncertainty is desirable. As such, many attempts have been made to unify probabilistic representations with a reasoning mechanism, amongst which, one of the key research areas of Artificial Intelligence has emerged known as *Probabilistic Logic Learning* (PLL) [45]. It investigates the possibility of integrating the probabilistic reasoning with first-order logic and machine learning. Probabilistic logic learning is also known as probabilistic inductive logic programming (PILP) which is an extension to ILP [105]. Stochastic Logic Programs (SLP), introduced by Stephen Muggleton in [88, 87], is one of PILP frameworks that generalizes logic programming in such a way that a clause can be associated with a probability.

SLPs contains a set of first-order range-restricted definite clauses, where each clause C can be associated with a label $p \in [0, 1]$ in the form of $p : C$. A definite clause is range-restricted if the term appearing in the consequent also appears in the antecedent. Note that the sum of associated labels should not exceed "1". In summary, arithmetic labels on clausal definitions are used by SLPs with the purpose of extending the logic programming [28]. Derivation mechanism employed by SLPs is *stochastic* Selection-function-Linear-resolution-for-Definite-clauses (SLD). Given a labeled goal clause $G \ p : G$ and a labeled clause $q : C$, stochastic SLD produces R , as the resolvent of G and C , in the form of $pq : R$. Failure Adjusted Maximization algorithm (FAM), implements parameter estimation of the probability labels associated with the clauses

in SLPs [40]. FAM implementation is provided as an open source software known as *Parameter estimation in Prolog (Pe-Pl)* [27, 40].

Stochastic logic programming is not in the scope of our project and hence, we did not intend to cover it in details, however, we performed some preliminary experiments with *Pe-Pl* in order to investigate the possibility of using probabilistic learning in the problems presented in this research. Those experiments will be discussed in chapter 6 (§7.4). For a brief, but very useful discussion on probabilistic inductive logic programming, we recommend [46], and for more details on the structure of SLP, refer to [89].

2.10 Conclusions

In this chapter we provided some background material to the subject of this research, where we investigate solving a type of AI problems that we will elaborate on throughout this thesis. We discussed knowledge representation and logical systems in general, and we described two of the main logical system representational schemes: propositional logic and first-order logic. A discussion of the differences between first-order and propositional logic was followed by describing logic programming and Prolog. We briefly discussed *automated reasoning* and tried to explain three main reasoning approaches that are relevant to our work: inductive, deductive and constraint reasoning.

Automated theorem proving (ATP) was discussed along with a summary on proofs by refutation and OTTER system. Constraint solving and its formal definition was also discussed. In addition, we argued why it might be fruitful to translate a problem into the syntax of various AI problem solving system. We briefly described a technique for expressing general problems in CSPs. Further, HR discovery system which is a combined reasoning system, performing Automated Theory Formation (ATF) was discussed. Some applications of HR were mentioned in addition to a brief explanation on how HR will be used in dynamic investigation problems.

We paid special attention to machine learning and data mining and their differences and clarified some main concepts of the popular Apriori algorithm. We also talked about a popular Java

based knowledge analysis suite known as Weka. Further, we elaborated on inductive reasoning and ILPs, and laid the foundations to a later discussion in chapter 9 on “downgrading” and “upgrading” of problem representations. Inverse entailment and Progol were also discussed along with a analysis on Progol’s search space. Finally, we tried to make a very short introduction to SLPs and probabilistic inductive logic programming as we envisage its use in our future work.

Chapter 3

Dynamic Investigation Problems

3.1 Introduction

One of the ultimate goals of AI computer programs is to solve real world problems as efficiently as, or even better than, people or to solve problems that cannot be solved by them. The purpose of this thesis, as mentioned in chapter 1, is to choose a set of problems, similar to real life investigation scenarios, to which we can employ various AI approaches, in concert, in order to solve them. To this purpose, we have largely focused on: (a) specifying a formal definition and explaining a type of hybrid AI problems which we have named “Dynamic Investigation Problem”, (b) automatically generating such problems and (c) developing methods for solving them. We designed such problems specifically to be more realistic than the kinds of problems solvable by standalone Artificial Intelligence approaches, such as theorem proving, machine learning or constraint solving. In overview, dynamic investigation problems are similar to real life police or medical investigations, i.e., a set of suspects are involved in the problem that can be presented as possible criminals or suspected illnesses. The problem contains facts and rules about a current investigation case and many other cases than can bear resemblance to the current case. Similar to real scenarios, the suspects, facts and/or rules of the investigation case can change at different times - hence the dynamic nature of the problem. The aim is to correctly rule out the bystanders and to identify the guilty party in the light of changing information. In

overview, we aim to design a prototype system that can aid doctors and/or detectives to solve medical or criminal mysteries.

In this chapter, we describe our motivations and the factors that inspired us to concentrate on dynamic investigation problems. With the help of an investigation puzzle from TPTP library [121], we explore the key aspects of these kinds of problems. We show how such problems can be tweaked in order to be amenable to machine learning, constraint solving and automated theorem proving. Further more, by removing a piece of information from the puzzle, while neither of the mentioned AI approaches can solve the problem, we show that a combination of AI techniques can solve it. Finally, we will present a formal definition of an Investigation Problem (IP) and a Dynamic Investigation Problem (DIP). In this chapter, we try to demonstrate DIPs through examples which are not necessarily true and are just for clarification purposes.

3.2 Motivations and Inspirations

Initially, we had to contend with choosing a type of AI problem akin to real life scenarios which we could tackle using different reasoning techniques. In this process we were inspired by the stories of Sherlock Holmes [4] and medical drama television series, House, M.D. [3]. In House stories, normally, a patient with a mysterious malady is presented to the protagonist Dr. House and his team. When all other doctors have lost hope on the patient, genius Dr. House with his astute logical reasoning takes the case and strives to solve it. During the diagnostics process, he tries to form a plausible theory which best explains patients symptoms. He then tries to prove each axiom of the theory by performing different medical tests. The interesting part is when throughout the diagnostics he encounters contradictory results. This makes him eliminate some of the symptoms as irrelevant or add unseen symptoms to the theory that he believes are implicitly embodied in the problem and may be manifested later on. Different reasoning approaches can be identified in his diagnostics process. Sometimes, he may reach a conclusion based on solid evidence, tests and rules he assumes to be true; this type of reasoning can be seen as deductive. For instance, he has two pieces of information which he assumes to be true

(i) the disease that has caused rashes is the main cause of nausea too and (ii) the only disease that could have caused rashes is flu. With the use of deductive reasoning, he infers that flu is the disease. On the other hand, Dr. House may sometimes face a dilemma which is to choose between two different diseases where both satisfy all the symptoms and curing both at the same type demands a risky procedure. He then runs new tests and finds out a new symptom by which he can rule out one of the diseases in that it does not satisfy the new symptom; in this case, House has used constraint based reasoning. Additionally, due to his years of experience and treating many patients, he knows that if a patient shows symptoms like *fever*, *rash* and *pain*, there is a great chance that *white blood cell counts* should elevate any time soon. Therefore, inductive reasoning has helped him identify a new piece of information. There could be cases where all the evidence supports a certain disease, however, symptoms are inconclusive. For instance, a patient has *nausea*, *movement imbalance* and *memory loss*. House knows that all the symptoms along with *cognitive impairment* can indicate *brain tumor*. Assuming that the patient has a brain tumor, he only needs to confirm *cognitive impairment* in order to treat for tumor. Therefore, using abductive reasoning he tests and proves the missing precondition that can cause brain tumor.

We first concentrated on scenarios in which the police investigates a murder case. In such scenarios, in addition to axiomatic background knowledge, there may be previous solved cases which bear resemblance to the current case. Given that the previous cases were solved, one can imagine employing a machine learning system to learn a set of rules which can classify suspects in the case as either guilty or not guilty. The rule set could then be applied to the current case. If only one person was classified as guilty, this would solve the problem. While this reasoning may not be sound, it would at least help to identify a *prime suspect*. In the current case, there may be information describing the particulars of the case, arising from physical evidence, motives, alibis, general knowledge, etc. If so, it may be possible to define a set of constraints that the guilty suspect must satisfy, and then use a constraint solver to rule out suspects. If only one suspect satisfies all the constraints, again the problem is solved. Alternatively, the same information about the case may be used as axioms in a theorem proving setting. In such a setting, one could attempt to prove a set of conjectures, each one stating that a particular

suspect is guilty/not guilty. If only one suspect is proved to be guilty (or alternately, it is possible to prove that all but one suspects are not guilty), then the problem is once again solved.

3.3 The Aunt Agatha Problem

As mentioned above, we aim to employ different approaches (machine learning, constraint solving and automated theorem proving) in order to solve dynamic investigation problems which we will formalize and explain shortly. To show the feasibility of using three different types of solvers to attack the same problem, we looked at the “Who Killed Aunt Agatha” problem from the TPTP library (i.e., problem PUZ001 [122], originally from [100]). We cherry-picked a “whodunnit” puzzle because it narrates a murder story in which the mystery is revealed by identifying the murderer. Such problems are akin to real life investigation scenarios we aim to tackle. Note that TPTP library contains a set of test problems for automated theorem proving (ATP) systems. The background knowledge for this problem, stated in English, as follows:

“Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, Butler and Charles live in Dreadbury Mansion and are the only people who live therein. A killer always hates the victim and is never richer than the victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the Butler. The Butler hates everyone not richer than Agatha. The Butler hates everyone Aunt Agatha hates. No one hates everyone and Agatha is not the Butler”.

This problem is usually posed as a logic puzzle for theorem provers, where the aim is to prove that Aunt Agatha killed herself. However, in a more general setting, the answer would not be given, therefore, we would be asked to find out who killed Aunt Agatha. With this tweak, we can make it amenable to all three different solving approaches mentioned above. It is worth mentioning that the solution process for this particular problem and its mutilated version will be discussed in more details in chapter 6.

3.3.1 Aunt Agatha – Machine Learning

To show that – in principle – investigation problems are amenable to machine learning, we first tried to tackle Aunt Agatha problem by applying a machine learning methodology. We invented some data which embodies the axioms of the problem. In particular, we wrote down the details of five case studies with three people in, one of whom had been murdered. We specified who was richer than who, who hated who, who was killed and who the murder was. This was done in such a way that:

- a) there was a direct mapping from Agatha, Butler and Charles to one of the people (suspects) in each of the case studies.
- b) all the axioms from the problem statement about who could possibly hate who, etc., were upheld by the suspects in the case studies. In other words, the same relations between Agatha, Butler and Charles were established between their corresponding suspects in the case studies.

In the first instance, the data reflected the fact that the murderer and the victim were always the same person – the Agatha character. This data was produced in the syntax of the Prolog machine learning system [84] and is depicted partially in figure 3.1 on page 56. We explained Prolog system and mode declarations in details in chapter 2 (§2.8.4). We ran Prolog and it hypothesized the rule that:

```
killer(A):- \+ victim(A).
```

Given that Prolog's output is generated in Prolog syntax, it was very easy to check that this profile applied to only Aunt Agatha in the current case. To make matters more interesting, in the second instance, we generated the data to still satisfy the axioms, but we varied the murderer/victim combination. In other words, the murderer was not the same as the victim in all the past cases. In this instance, Prolog hypothesized the following rule in FOL format:

```
killer(A,B):- hates(A,B), \+ richer(A,B).
```

```

:- modeh(1, killer(+person))?
:- modeb(1, victim(+person))?
:- modeb(1, hates(+person, #person))?
:- modeb(1, richer(+person, #person))?

% Case 1
person(sophia).
person(joe).
person(kate).
hates(sophia, sophia).
hates(sophia, kate).
:- hates(kate, sophia).
richer(sophia, joe).
victim(sophia).
:- killer(joe).
killer(andrea).

```

Figure 3.1: Aunt Agatha: part of a case study

When we applied the above rule to the current case, only Aunt Agatha fitted the profile. Based on the above results, by handcrafting the data, Progol was able to find the underlying axioms from an automated theorem proving problem. This sparked an idea that it is possible to use Progol to learn the profile of the guilty party and use those axioms to solve the current case mystery.

3.3.2 Aunt Agatha – Constraint Solving

To show that such problems are amenable to a constraint solving approach, we wrote a constraint satisfaction problem (CSP) in the syntax of the SICStus Prolog CLPFD module [21]. We defined the problem as a CSP with one variable which could take one of three values representing Agatha, Butler and Charles respectively and was constrained as per the axioms of the problem. Therefore, the domain of the variable comprised these three people that can be referred to as *suspects*. As for the constraints, two of the problem axioms *killer always hates the victim* and *killer is never richer than the victim* were instantiated. The rest of the axioms were regarded as background information. Part of the problem in SICStus Prolog syntax as shown in figure 3.2.

```

pseudonym(1, 'Aunt Agatha').
pseudonym(2, 'Butler').
pseudonym(3, 'Charles').
killed_agatha(X):-
    domain([X],1,3),
    killed(X,1),
    labeling([], [X]).

```

Figure 3.2: Part of Aunt Agatha problem in Prolog syntax

We made 1, 2 and 3 adopt one of the suspects' names with the use of “pseudonym”, as CLPFD allows only integer domains. `labeling(Options, +Variables)` [50], tries to assign variables to a value in their domain and indicates that the posted constraint is satisfied in case a true assignment can be found. The order of the variables listed for assignment can be controlled using the “Options”.

The guilty suspect(s) should satisfy the constraints for being the murderer and the rest of the suspects should be ruled out. Following statements depict the output of the CSP solver showing that SICStus solved the problem by ruling out Butler and Charles and not Agatha, hence, the problem was solved.

Aunt Agatha has not been ruled out.
Charles cannot be the killer.
Butler cannot be the killer.

3.3.3 Aunt Agatha – Automated Theorem Proving

Finally, in order to show such whodunnit problems are amenable to solution by theorem provers, we specified six conjectures to the OTTER theorem prover [79]. We explained automated theorem proving and OTTER in chapter 2 (§2.4 and §2.4.1). Part of the conjectures representing the information from the problem statement is shown in figure 3.3.

The conjectures to be proved were respectively: Agatha killed/didn't kill Agatha; Butler killed/didn't kill Agatha; Charles killed/didn't kill Agatha. OTTER successfully proved that Agatha killed Agatha and Butler and Charles didn't kill Agatha. It failed to prove any other conjectures, hence the problem was solved.

$all\ x\ (\neg(richer(x, agatha)) \rightarrow hates(butler, x)).$
 $all\ x\ (hates(agatha, x) \rightarrow hates(butler, x)).$
 $all\ x\ (hates(agatha, x) \rightarrow \neg(hates(charles, x))).$

Figure 3.3: Part of the input to OTTER

3.3.4 Mutilated Aunt Agatha

As we mentioned in section §3.1, in real world scenarios, there may not be enough background information for any single solver to solve the problem and we are interested in cases where it may be possible to combine a machine learning, theorem proving and constraint solving approaches in order to best use the obscure background information of cases in order to solve them. The Aunt Agatha problem becomes more interesting if we remove information from each of the three problem statements in such a way that neither Progol, SICStus nor OTTER can solve the problem. We removed the following axiom from the theorem proving and CSP problems:

$$\forall X(person(X) \rightarrow \exists Y(person(Y) \wedge \neg hates(X, Y))).$$

The axiom states that for every person X , there is a person Y , who X does not hate him which can be rewritten as:

$$\neg(\exists X(person(X) \rightarrow \forall Y(person(Y) \wedge hates(X, Y)))).$$

Above axiom is the translation of “no one hates everyone” in the background knowledge and is crucial to solving the problem. The reason we rewrote the axiom is because HR can later recover this axiom in both forms. Without this axiom, SICStus could not rule out Butler as the killer and OTTER proved that Butler and Agatha both killed Agatha, hence neither of the methodologies worked. We investigated whether the data from the machine learning approach could be used to recover the missing axiom. As explained earlier in §3.3.1, we already invented a set of case studies where the axioms from the problem statement were upheld by the suspects in each of the cases. In particular, we employed the HR automated theory formation [29], explained in details in chapter 2 (§2.6), to form a theory about the previous case studies. Using

HR's *forall*, *exists*, *negate* and *compose* production rules, HR made the conjecture that in all case studies:

$$\nexists x \quad s.t. \quad person(x) \wedge (\forall y, (person(y) \rightarrow hates(x, y))).$$

This states that, in all cases, there is no person who hates everyone. Hence, we realized that HR can recover the missing axiom. By feeding the recovered axiom to constraint solver and theorem prover we consequently solved the problem by both solvers. We will not elaborate on the solution details in this chapter, but suffice to say, we performed a series of experiments with Aunt Agatha problem in order to show that such problems are amenable to either of the mentioned methodologies. We will thoroughly explain in chapters 6 and 7, how HR, OTTER and other methodologies can be harnessed to solve Aunt Agatha and DIP problems in general.

3.4 Investigation Problem (IP) - Formal Definition

Before defining IPs, it is essential to remind the l -connectedness definition from [30] as follows:

Definition 3.1. Suppose C is a clause of the form mentioned in formula 3.1. Each X_i is a

$$P(X_1, \dots, X_m) : - \quad P_1(Y_{11}, \dots, Y_{1n_1}), \dots, P_l(Y_{l1}, \dots, Y_{ln_l}) \quad (3.1)$$

variable and each Y_{ij} may be a variable possibly the same as X_i or a ground term. Then the variable V which is a literal in the body of C is said to be l -connected if it upholds the recursive conditions shown in figure 3.4.

- $V = X_1$ or
- $\exists i, j, k \quad s.t. \quad j \neq k, Y_{ij} = V \quad \text{and} \quad Y_{ik} = X_1$ or
- $\exists i, j, k \quad s.t. \quad j \neq k, Y_{ij} = V \quad \text{and} \quad Y_{ik}$ is a l -connected variable

Figure 3.4: Conditions of l -connectedness

We consider investigation problems as being similar to CSPs with a finite set of variables, each associated with a finite domain and a set of constraints. Recalling from the previous section §3.3,

we tweaked Aunt Agatha automated theorem proving problem in such a way that it became amenable to a combination of machine learning and constraint solving methodologies. For this purpose, we invented data in the form of case studies similar to the current case scenario and fed the data to Progol machine learning system in order to learn the profile of the guilty party. With that tweak, the resulting Aunt Agatha problem was different to the original ATP problem as it contained previous case studies in one instance to fit the machine learning and a set of rules identified as constraints to fit the constraint solver. Proceeding with our experimentations on such problems, we mutilated Aunt Agatha in order to investigate the possibility of solving the problem in case a piece of information is missing. We recovered the crucial axiom that neither of the mentioned methodologies could solve the problem in its absence. The intention was to investigate if we can solve real world medical or murder mysteries in which some piece of information maybe missing or at least not explicit enough to help solving the problem; such medical or criminal investigation problems may be analogous to a set of solved cases that can come handy in solving the current mystery. Inspired by that idea, the investigation problem (IP) that we are intended to formalize has two sections: (a) *Current Case* and (b) *Past Cases*. In an IP, unlike standard CSPs, there might not be a single or a set of *definite* answer(s), yet there might be a set of inconclusive answers where each carry a likelihood degree; the likelihood degrees indicate how confident we can be in considering them as answers. Hence, an IP can be similar to a partial constraint satisfaction problem (PCSP) introduced in [127] and explained in chapter 2 (§2.5.1).

3.4.1 IP: Current Case

As mentioned earlier, an investigation problem is comprised of a current case and a set of past cases. The proceeding definitions are intended to formalize the current case of an IP. We should bear in mind that each of the past cases follow the same formalization as they are constituents of an IP similar to the current case. There are only a few differences between current and past cases that will be mentioned in the following sections.

Definition 3.2. Let S be a set of suspects:

$$S = \{s_1, s_2, \dots, s_k\} \quad \text{such that} \quad \forall s_i \in S : \quad s_i \quad \text{is a suspect} \quad (3.2)$$

S is a set of suspects that constitute the **answer** to our investigation problem. It is obvious that the answers are not included in the problem description of the current case and should be discovered through the solution process. In every IPs and later DIP, the answer is a set that can contain more than one suspect, hence S can have more than one member. In other words, S can be seen as the domain of **answer** and $\{s_1, s_2, \dots, s_k\}$ are domain values. The domain of each answer is a set of all possible values that can be assigned to each answer. We will explain the *answer* to an IP and/or DIP, in detail, in chapter 7 (§7.3). For time being, let us consider the *answer*.

Definition 3.3. Let A be the set of answers:

$$A = \{A_1, A_2, \dots, A_m\}$$

where m is the number of AI systems involved in the solution process

$$\forall A_i \in A : \quad A_i = \{x_1, \dots, x_k\}$$

$$\forall x_k \in A_i : \quad D_{x_i} \text{ is the domain of } x_k \text{ in } A_i \quad \text{such that} \quad D_{x_i} \subseteq S \quad (3.3)$$

D_x is the domain of x_i contains all the values that can be assigned to x_i for the answer A_i . The reason D_{x_i} is a subset of S and not always the same as S , is due to the nature of investigation problems; the number of suspects involved in the mystery can change over time. A few of the suspects maybe ruled out and some more may be added with the light of new evidence. This part is relevant to DIPs and will be explained shortly. By solving an IP/DIP we mean that the values from S are assigned to each x_i in A_i . In addition, we explained in §3.1 that a combination of reasoning systems will be harnessed to play in concert in order to solve an IP/DIP. Each of these combination of systems that will be explained in chapter 7 (§7.3), will generate a single solution, hence A contains a set of answers.

Definition 3.4. Let P be a set of facts:

P is a herbrand set, consisting of all ground terms. Each $P_i \in P$ is a predicate constituting the

$$P = \{P_1, \dots, P_k\} \quad \text{where } k \text{ is an arbitrary number} \quad (3.4)$$

background knowledge of the problem. We should bear in mind that each predicate can appear more than once in the problem description. We should note that the *negation* of each predicate may also appear in the background knowledge. Each predicate P represents a relation between a set of suspects or simply declares a **fact** about a suspect(s). Each predicate contains a set of *suspects* and *constants* and is of finite arity. These predicates can appear in the rules that will be defined in the next definition, however, they can contain variables as well.

Definition 3.5. Let C be a set of definite clauses (Horn clauses with exactly one positive literal):

$$C = \{C_1, C_2, \dots, C_m\} \quad (3.5)$$

C is a set of definite clauses where each of those clauses can be identified as **rules** presented in the problem description. Every clause C_i contains predicates and variables in the form of:

$$C_i(X) : - \quad P_{i1}(Y_{11}, \dots, Y_{1n}), \dots, P_{ia}(Y_{a1}, \dots, Y_{al}) \quad (3.6)$$

where:

- a, l, n are arbitrary finite values
- The arguments in P_{ij} may each be a variable or a ground term.
- Clause C_i is l-connected

Each of these rules can be seen as **constraints** that should be satisfied by the suspects in order to fit the domain of answers. We are interested in solutions to IPs which are an ordered list of domain values for X . In other words, every suspect should satisfy the rules defined in the problem. These rules are either demonstrating the profile of the guilty suspect or are the part of the constraints that should be satisfied by the suspects appearing in the answer. We described “constraints” in chapter 2 (§2.5) and constraint translation in §2.5.2. We will discuss

how we reformulate each of these rules in the format of SICStus CLPFD constraints in chapter 6 (§6.2.4). It is also worth mentioning that $Pred_{C_i}$ is a set of predicates that appear in the clause C_i , hence $Pred_{C_i} \subseteq P$.

Definition 3.6. Considering the definition 3.3, let:

$$\begin{aligned} S' &= [x'_1, x'_2, \dots, x'_k] \quad \text{where} \\ x'_i &\text{ is the likelihood degree of } x_i \text{ and } x_i \in D_x \end{aligned} \quad (3.7)$$

The likelihood degree x'_i indicates how likely the suspect x_i can fit the IP's answer.

$$\text{likelihood degree of } x_i = \frac{\text{Number of rules satisfied by } x_i}{\text{Total number of rules}} \quad (3.8)$$

The rules being satisfied by x'_i are either the rules demonstrated in the problem description or from the past cases' embedded rules. We will elaborate on the past cases rules shortly in §3.4.2. The more rules each domain value satisfies, the higher likelihood it has for being the correct answer, thus the answers can be ordered. In summary, the current case of an IP:

$$\text{Current Case of IP} : \langle \text{suspects}, \text{constants}, \text{facts}, \text{rules} \rangle \quad (3.9)$$

3.4.2 IP: Past Cases

As mentioned in the former section, an investigation problem is comprised of a current case and a set of past cases. The current case of an IP was formalized in the previous section and we also noted that the past cases follow the same formalization as they are constituents of an IP similar to the current case. Here, we try to formalize the additional factors involved in the past cases.

Definition 3.7. Let E be a set of past cases:

$$E = \{E_1, E_2, \dots, E_p\} \quad \text{where } p \text{ is an arbitrary number}$$

$$\forall E_i \in E, \quad E_i : < \text{suspects}, \text{constants}, \text{facts}, \text{rules}, \text{mappings}, \text{answer}(s) > \quad (3.10)$$

Each E_i is comprised of all the factors defined in the previous section. In other words, every E_i is similar to current case, whereas the difference is the addition *mappings* and *answer*. As mentioned earlier in §3.3.1, one of the important factors in solving an IP/DIP is to learn the profile of the guilty party with the help of previously solved past cases; therefore, each past case should contain an answer. In addition, we mentioned the past cases should bear resemblance to the current case similar to an investigation problem where a detective or a medical doctor's decision can be influence by the evidences he/she has already collected from the other cases. In overview, in every past case there is an analogy between its constituents and their corresponding factors in the current case, hence, the mappings. We will explain the mapping function shortly in §3.4.3, but suffice to say that it is needed to be sure about the similarity between the current case and each of the past cases. For instance, the predicates appearing in the facts and/or rules should have the same number of arities in both current and past cases.

Definition 3.8. We define a set of entities that will be helpful mostly during the IP/DIP automatic generation, discussed in chapter 4. Let:

$$Pred_{E_i} : \text{be the set of predicates appearing in the past case } E_i \quad (3.11)$$

$$Pred_E : \text{be the union of all the predicates present in all of the past cases} \quad (3.12)$$

$$A_{E_i} : \text{consists of ground unit clauses of a single target predicate known as “answer”} \quad (3.13)$$

We should bear in mind that the ground values in the target predicate “**answer**” in A_{E_i} are the suspects identified as the answers to E_i .

3.4.3 Definition of an IP

As mentioned in the previous sections, an IP contains a *current case* and a set of *past cases*.

IP satisfies the following conditions:

- (i) There should be an overlap between the predicates present in every past case and the overall predicates appearing in the current case. Considering the set of predicates P identified as facts in the current case, mentioned in definition 3.4, the commonality confirms the relevance between a past case and the current case. There is a subset of facts F in every E_i that also appears in the current case. For instance, in all the past cases and the current case, there is a suspect that is angry.

$$\forall E_i, \exists F \subseteq \text{Pred}_{E_i} \text{ such that } F \subseteq P \quad (3.14)$$

- (ii) There should be an analogy between all of the past cases and the current case so that we can correspond between at least a suspect in each of the past cases and the current case. Recalling from chapter 2 (§2.3), (Δ, Θ) model is the representative of the semantics of an n-ary predicate with Δ as the domain and Θ as the interpretations. Considering this piece of information and also F in the equation 3.14:

$$\begin{aligned} &(\Delta_1, \Theta_1) \text{ representing } f_1 \in F \\ &(\Delta_2, \Theta_2) \text{ representing } f_1 \in P \\ &\forall f_1 \in F, \exists M : (\Delta_1, \Theta_1) \rightarrow (\Delta_2, \Theta_2) \text{ where} \\ &M \text{ is a mapping function} \end{aligned} \quad (3.15)$$

A note on Mapping Function: M is a mapping function. Let $X : \langle x_1, x_2, \dots, x_n \rangle$ be the set of suspects in the current case and $Y : \langle y_{11}, y_{12}, \dots, y_{1n} \rangle$ be the set of suspects in the past case E_1 . Mapping function M associates suspects in the past case to suspects in the current case or maps suspects from X into Y . For instance, given $(x_1 : y_{11})$ in an IP/DIP, we know that y_{11} represents x_1 in the past case E_1 and bears the

same characteristics of x_1 . By characteristics, we mean y_{11} holds the same relationships with other suspects incorporated in the same predicates. Figure 3.5 depicts the bijection mapping where every suspect in X is mapped into a suspect in Y .

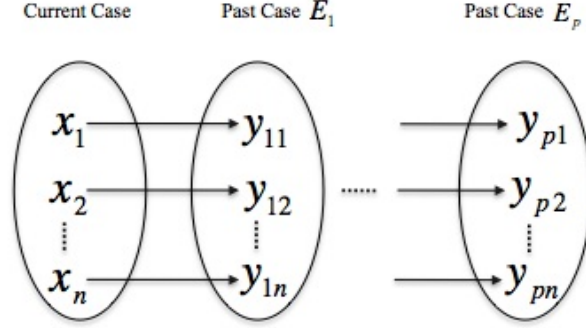


Figure 3.5: Mapping function $M : CurrentCase \rightarrow PastCases$

In general, an IP boils down to:

$$IP :< suspects, constants, facts, rules, case studies, mappings > \quad (3.16)$$

3.4.4 Isomorphic IP

As mentioned earlier, past cases similar to real life investigation scenarios, are a set of solved cases each containing at least a fact and a suspect that can be mapped to the current case according to §3.4.3. Isomorphic IP is a special case of IP in which the past cases are exactly the same as the current case except they are solved cases. This way, every extracted information from the past cases through machine learning techniques can be applied to the current case as past cases exhibit the identical characteristics to the current case. Therefore, these scenarios are the easiest to solve. Isomorphic IP satisfies the following condition:

(i) Using the equation 3.15:

$$\begin{aligned} \forall f \in F, \quad \exists P_i \in P \text{ and } \exists M : F \rightarrow P \quad \text{such that} \\ \Theta_f = \Theta_{P_i} \end{aligned} \tag{3.17}$$

M is a bijective mapping function. In other words, it is possible to map every fact in a past case to a unique fact in the current case and vice versa.

3.5 Definition of a Dynamic Investigation Problem (DIP)

A dynamic investigation problem is an IP with additional time aspect in the problem. In particular, the background knowledge keeps changing over time. Imagine a detective or a diagnostician who is initially presented with a set of information about a problem. As time passes, he/she discovers more information about the case – hence, there is dynamic nature to the problem. With the change in time, past cases, current case and the domain of the variables can be modified.

Definition 3.9. Let $T = \{t_1, t_2, \dots, t_n\}$ be the set of time instances. Using the definition mentioned in the former sections (definitions: 3.2, 3.4, 3.5), at each time instance, DIP constituents may change as follows:

$$\begin{aligned} \text{at } T = t_1, \quad IP_1 : < S_{IP_1}, P_{IP_1}, C_{IP_1}, case_studies_{IP_1}, mapping_{IP_1} > \\ & \vdots \\ \text{at } T = t_n, \quad IP_n : < S_{IP_n}, P_{IP_n}, C_{IP_n}, case_studies_{IP_n}, mapping_{IP_n} > \end{aligned}$$

We should bear in mind that each $IP_i (1 \leq i \leq n)$ satisfies the conditions for an IP. Over time, suspects, constants, facts and rules can be altered and, subsequently, the case studies.

Nonetheless, the following condition should be satisfied:

$$\begin{aligned}
& \exists s \in S_{IP_1} \text{ such that } s \in S_{IP_2} \wedge \cdots \wedge s \in S_{IP_n} \\
& \exists p \in P_{IP_1} \text{ such that } p \in P_{IP_2} \wedge \cdots \wedge p \in P_{IP_n} \\
& \exists c \in C_{IP_1} \text{ such that } c \in C_{IP_2} \wedge \cdots \wedge c \in C_{IP_n}
\end{aligned} \tag{3.18}$$

The condition shown in formula 3.18 implies that even though the background changes at each time instance, there is always an intersection between the constituents of the IPs. Due to the change of suspect domains, rules and facts, the answers at each time instance will be changed accordingly. Considering the definitions 3.3 and 3.6, the number of rules that can be satisfied by the suspect domain may change due to the change in rules, hence the change in likelihood degrees. It is possible to achieve a more accurate likelihood degree for each suspect once the investigation process is towards the end or terminated. We will see the results of the solution process in chapter 9. This is due to the augmenting information obtained through time steps. We should also bear in mind that in a dynamic investigation problem, at initial time steps, there may not be enough information to be able to correlate with a past case. In other words, during the initial steps, it might be harder to discover any useful information from the past cases.

3.6 DIP: a toy example

A simple toy example of a DIP in medical domain is shown in this section. The example is designed in such a way that it will be solved in three time steps where in each time step, new pieces of information are revealed. The definitions introduced throughout this chapter will be used to convey a better understanding of the example constituents as well as definitions themselves.

Stage 1:

Suppose an old patient p_0 has been admitted to a hospital with acute vomiting, he can walk

but exhibits partial paralysis. The patient is also feverish. The doctor is going to diagnose his disease based on the current symptoms. The set of suspected illnesses at each time step is S .

at $T = t_1$, IP_1 :

current case: patient p_0 , set of suspects $S_{IP_1} : \{old_age, heart_disease, swine_flu, polio, cancer\}$

set of facts P_{IP_1} :

```
symptom(vomiting,p0).
symptom(deafness,p0).
symptom(paralysis,p0).
symptom(fever,p0).
cause_of(vomiting, swine_flu).
cause_of(paralysis, heart_disease).
cause_of(paralysis, polio).
cause_of(vomiting, polio).
```

set of rules C_{IP_1} (includes C_1 and C_2):

```
disease(Diagnosis,Patient):- symptom(vomiting, Patient),
                                cause_of(vomiting, Diagnosis).
disease(Diagnosis,Patient):- symptom(paralysis, Patient),
                                cause_of(paralysis, Diagnosis).
```

set of past cases at $T = t_1$, IP_1 :

case 1: E_1 : patient p_1 , set of suspects $S_1 = \{cancer, swine_flu\}$

```
symptom(fever, p1).
symptom(cough, p1).
symptom(rash, p1).
disease(swine_flu, p1).
```

case 2: E_2 : patient p_2 , set of suspects $S_2 = \{swine_flu, polio\}$

```
symptom(blindness, p2).
symptom(paralysis, p2).
symptom(fever, p2).
disease(swine_flu, p2).
```

case 3: E_3 : patient p_3 , set of suspects $S_3 = \{cancer, swine_flu, polio\}$

```

symptom(paralysis, p3).
\+ symptom(blindness, p3).
symptom(vomiting, p3).
symptom(fever, p3).
disease(swine_flu, p3).

```

solution at $T = t_1$:

The doctor initially believes that only rules in C_{IP_1} , (C_1 and C_2) can determine the disease. Based on the facts mentioned in P_{IP_1} , the doctor knows that Polio satisfies C_1 as well as C_2 , whereas heart disease satisfies C_2 and Swine flu only satisfies C_1 . Therefore, according to the likelihood degree formula 3.7, Polio, satisfying both existing rules, holds the highest likelihood degree 100% and heart disease and swine flu each 50% likely to be the guilty suspects. Based on the facts and rules mentioned in the current case, the doctor can prioritize the possible diseases as shown in table 3.1.

Priority	Disease	C_1	C_2
1	polio	✓	✓
2	heart disease	✗	✓
2	swine flu	✓	✗
3	old age	✗	✗
3	cancer	✗	✗

Table 3.1: Answer: $T = t_1$, rules: C_1, C_2

Note: the symbols in the tables indicate: ✓: satisfying ✗: not satisfying

Additionally, based on the past cases, the doctor can infer that a patient can have swine flu in case of having fever. Considering this rule based on the past cases, a new piece of information is revealed: it is now known that fever can be explained by swine flu. Therefore, the diagnosis holds a new rule as:

```

 $C_3$  : disease(swine_flu,P):- symptom(fever,P).

```

The new ordering is changed according to table 3.2

Stage 2:

Later on, the doctor finds out that the patient has high blood pressure and his respiratory system is about to fail in addition to his previous symptoms. Moreover, the doctor already

Priority	Disease	C ₁	C ₂	C ₃
1	polio	✓	✓	✗
1	swine flu	✓	✗	✓
2	heart disease	✗	✓	✗
3	old age	✗	✗	✗
3	cancer	✗	✗	✗

Table 3.2: Answer: $T = t_1$, rules: C_1, C_2, C_3

knows that cancer can be the cause for high blood pressure and vomiting.

at $T = t_2$, IP_2 :

patient p_0 , set of suspects $S_{IP_2} : \{cancer, old_age, heart_disease, swine_flu, polio\}$

set of facts P_{IP_2} : (the following facts are added to the previous P_{IP_1} facts)

```
symptom(high_blood_pressure,p0).
symptom(respiratory_failure,p0).
cause_of(high_blood_pressure, cancer).
cause_of(vomiting, cancer).
```

set of rules C_{IP_2} (C_4) :

```
disease(Diagnosis, Patient):- symptom(vomiting, Patient),
                               symptom(high\_blood\_pressure, Patient).
```

solution at $T = t_2$:

At the stage, according to rule C_4 , the guilty suspect (disease) should explain vomiting and high blood pressure where only cancer fits this profile. Considering facts, rules and suspected illnesses, agglomerated from t_1 and t_2 , the new ordering is shown in table 3.3.

Priority	Disease	C ₁	C ₂	C ₃	C ₄
1	polio	✓	✓	✗	✗
1	cancer	✓	✗	✗	✓
1	swine flu	✓	✗	✓	✗
2	heart disease	✗	✓	✗	✗
3	old age	✗	✗	✗	✗

Table 3.3: Answer: $T = t_2$, rules: C_1, C_2, C_3, C_4

set of past cases at $T = t_2$, IP_2 :

case 1: E_1 : patient p_4 , set of suspects $S_4 = \{cancer, swine_flu\}$

```

symptom(respiratory_failure, p4).
symptom(high_blood_pressure, p4)
symptom(weakness, p4).
symptom(paralysis, p4).
disease(cancer, p4).

```

case 2: E_2 : patient p_5 , set of suspects $S_5 = \{polio, cancer\}$

```

symptom(high_blood_pressure, p5).
symptom(respiratory_failure, p5)
symptom(fever, p5).
disease(cancer, p5).

```

case 3: E_3 : patient p_6 , set of suspects $S_6 = \{cancer, polio\}$

```

symptom(vomiting, p6).
symptom(respiratory_failure, p6).
symptom(rash, p6).
disease(cancer, p6).

```

Here, the doctor can infer based on the similar solved cases, that every cancer patient should be having respiratory failure. Therefore, the recent past cases reveal a new rule as:

C_5 : `disease(cancer, P) :- symptom(respiratory_failure, P).`

According to the set of facts P_{IP_2} , mentioned earlier, the patient's respiratory system was about to fail; considering this fact and given the rule C_5 , the new ordering is as shown in table 3.4.

Priority	Disease	C_1	C_2	C_3	C_4	C_5
1	cancer	✓	✗	✗	✓	✓
2	polio	✓	✓	✗	✗	✗
2	swine flu	✓	✗	✓	✗	✗
3	heart disease	✗	✓	✗	✗	✗
4	old age	✗	✗	✗	✗	✗

Table 3.4: Answer: $T = t_2$, rules: C_1, C_2, C_3, C_4, C_5

It is clear from the case studies throughout this toy example that there always exists an intersection between the suspects in the *current case* and all the *past cases*. This satisfies the DIP condition mentioned in formula 3.18. The change in the likelihood degree of suspects is shown

with the help of the “answer tables”. In other words, throughout the solution process, the ordering of the suspects changes; table 3.1 shows that Polio is most likely to be the main cause of the patient’s symptoms. However, in the light of information gathered through case studies and time steps, we can see that Polio and swine flu both hold the same likelihood degree in table 3.2; with the help of information extracted from case studies at time t_2 , cancer attracts more suspicion as table 3.3 depicts. Ultimately cancer becomes the suspect explaining most of the symptoms and Polio and swine flu come second as shown in table 3.4. With the help of this toy example, we aimed to demonstrate how a DIP information, similar to a real life investigation scenarios, changes and hence the change in answers. Throughout this thesis we aim to generate DIPs with various levels of difficulty and to employ a combination of reasoning techniques in order to solve them.

3.7 Conclusions

As mentioned in the introduction section §3.1, our goal was to explore and to formally define a certain type of AI problem similar to real life investigation scenarios and to tackle them by employing a combination of reasoning techniques. Having considered stories of Sherlock Holmes and medical drama House, M.D. we tried to build and solve similar toy problems and we narrated them in such a way to be amenable to a combination of machine learning, constraint solving and theorem proving techniques. In section §3.3, we discussed that using only the data about the current case, these problems can be modeled most easily as an automated theorem proving problem and/or constraint satisfaction problem which has a single solution. However, this does not take into account the case studies, within which pertinent information about the current case may be hidden. On the other hand, using only the case studies, these problems can be modeled most easily as a machine learning problem, from which a profile of the guilty party can be learned. The profile can be applied to the current suspects, and the one most fitting the bill is the guilty one. From generalizing such problems, we formally defined investigation problems (IP) and dynamic investigation problems (DIP). Dynamic investigation problems are hybrid problems, and as such are more realistic and of interest to the wider AI community.

Our first contribution on this project was to formally define investigation and dynamic investigation problems in terms of existing terminology relating to machine learning and constraint solving and automated theorem proving. In the following chapters, we will try to build a system capable of automatically generating dynamic investigation problems, and we experiment with its usage to produce more varied problem sets in terms of difficulty.

Chapter 4

Automatic Generation of Dynamic Investigation Problems

4.1 The GH System – Introduction

As mentioned in chapter 3 (§3.2), we aimed to choose certain AI problems akin to real life scenarios which we could tackle using different reasoning techniques. Based on that, we explored and formally defined “Investigation Problems” (IP) in §3.4 and “Dynamic Investigation Problems” (DIP) in §3.5. We showed in chapter 3 that such problems are amenable to a combination of machine learning, constraint solving and automated theorem proving. We also showed through examples that a combination of reasoning systems can be applied in order to help find some missing information or best use of obfuscated background knowledge.

In order to explore the methods of solving DIPs, we first generate them systematically. This is because through generation, we can create scenarios with various difficulties that can ultimately gauge the efficiency of solving methods. Bear in mind that a DIP should use an *interlingua* or an automated translator so that the problem description can be understandable by different reasoning systems we aim to use in the solution process. We describe the generation process in three steps : (a) defining the difficulty level and specifying the correct answer(s) along with the their likelihood degrees according to the formula 3.7 in chapter 3 (b) generating a problem

(c) translating the problem into syntaxes of various AI systems that are intended to be used to find a solution.

Automatically generating DIPs, translating and ultimately solving them using a combination of reasoning techniques are constituents of a system which we have built and will explain throughout this thesis. We have named this system after the protagonist of the House, M.D. television series – Gregory House – as GH. We mentioned in §3.2 that stories of Sherlock Holmes and House, M.D were both inspirational to our work and since we needed only one name, we picked Gregory House (GH).

In this chapter, we will discuss the method that GH employs to automatically generate dynamic investigation problems based on the formal definitions in §3.5. This method was first introduced by the author of this thesis in [106]. We start by generating a board game known as Cluedo. Furthermore, we introduce an algorithm for generating a generic type of DIP. We will also show the methods we used in order to translate DIPs into the syntax of CLPFD (discussed in §2.5.1), Progol (§2.8.4) and HR (§2.6) and finally, we describe a technique we developed to restructure a DIP into Weka (§2.7.3) format.

4.2 Representation language

During the generation process, choosing the language in which a DIP should be represented is a key factor. In order to use different reasoning techniques to tackle a DIP, we need to translate every generated DIP to the syntaxes of AI systems involved. Therefore, expressing a DIP in such a way that it would ease the translation process is essential.

First order predicate logic is both precise and expressive enough to define our problems with it. In addition, as we see later, we aim to use CLPFD module of SICStus [21] as constraint solver and Progol [84] as one of our machine learning approaches. As Progol is a first order learning system, representing DIPs in SICStus Prolog [50] syntax will give us the ability to easily translate them to Progol syntax. Also, translating to SICStus CLPFD would be without any major difficulty.

4.3 Automatic generation of Cluedo – style Problems

To show that a dynamic investigation problem can be automatically generated according to its formal definition, we initially considered a board game known as Cluedo [2] which will ultimately pave the way for generating generic type of DIPs. As explained in chapter 3 (§3.3), we chose Cluedo puzzle because it narrates a murder story in which the mystery is revealed by identifying the murderer and is akin to real life investigation scenarios we aim to tackle. In this game the player moves around a mansion with nine rooms where the murder can take place and collects clues to infer which suspect has murdered the victim. In the classic single player Cluedo game, the player tries to determine the identity of the murderer by searching every single room. We can draw an analogy between each room and a time step in our DIP definition. Therefore, the information gathered by the player at each room provides the facts, suspects and rules of the DIP at time T . The program should randomly generate different scenarios and output a limited set of rules and predicates at each time slot. As explained in the previous chapter’s toy example (§3.6), the facts and rules can help identifying the guilty suspects. Hence, the output of the generative process is similar to a real game in which the player can collect few clues at every room. To make the game more interesting, the amount of information being generated at each step is also varied. Hence, the player may find more information in one room and less in another. Additionally, we increased the complexity of the problem by adding rules and predicates about the suspects and the murder case in general. Even though we try to randomly generate Cluedo scenarios, we make the problem more general and complex by adding more data. For instance “*the murderer should be tall and quick*” or “*the murderer should be angry with someone*” are added as rules (constraints). The extra predicates such as “*Professor Plum is tall*” or “*Scarlett is short*” can be complimentary to the constraints.

As mentioned in chapter §3, if the information about the current case is not sufficient to solve the mystery, previously solved cases can hopefully help us learn the profile of the guilty party. For instance, *the murderer should be tall and left-handed*. These rules can be applied to the current case suspects and solve the problem or at least make the problem easier to be solved. Therefore, a DIP consists of a current case and past cases that will be generated simultaneously.

In this section we will explain the generation algorithm of a more complicated Cluedo board game. Furthermore, we will discuss what we mean by a *generic type* and we will introduce the generation algorithm for generic DIPs in the next section.

Solving DIPs will be explored later in the chapters 6 and 7 and we will elaborate on how to gauge the difficulty of a problem. However, it is worth mentioning that the aim is not to generate a DIP that can be easily solved. To clarify the word “easily”, we can think of generating DIPs in two modes as follows, where the latter is in our intention.

- **Default mode** is the mode in which the background information is sufficient enough to be solved by any standalone AI approaches we mentioned in chapter 3. In other words, the current case of the DIP can be “*easily*” solved by a CSP solver or a machine learning system or an automated theorem prover alone. Indeed, it is not worth to generate such problems as our purpose is to employ a combination of various reasoning techniques for solving problems with ill-formed background knowledge. We only used the default mode during our preliminary analysis which helped us develop a systematic way of complicating DIPs and generating the real mode discussed below.
- **Real mode** is the mode in which neither machine learning, constraint solving nor theorem proving techniques, alone, can solve the problem. Real mode problems have at least one of the following properties:
 - The key bits of information required for solving the mystery are embodied in such a way that are not clear from the problem description. For instance, the constraint declaring that the “*murderer should be tall and angry*” is not explicitly declared.
 - Some pieces of information, crucial for solving the problem, are missing. For instance, “*professor Plum loves Scarlett*” is missing given “*the murderer should be in love with Scarlett*”.

Based on the above categorization, we generate Cluedo scenarios in the *real mode*, as per algorithm 4.1 in figure 4.1. We should note that the terminologies mentioned in the algorithm are according to IP/DIP constituents mentioned in chapter 3 (§3.4).

Algorithm 4.1: Cluedo Generation

```

input:  $n(C)$  = number of rules
input:  $C = \{C_1, C_2, \dots, C_c\}$  set of constraints, Ex:  $C_1 : \text{angry}(X) :- \text{hates}(X, Y) .$ 
input:  $C_{\text{murderer}}$  Ex:  $\text{murderer}(X) :- \text{angry}(X), \backslash + \text{richer}(X, Y) .$ 
input:  $n(P)$  = number of facts (predicates)
input:  $P = \{P_1, P_2, \dots, P_p\}$  set of facts
input:  $n(S)$  = number of suspects
input:  $S$  = set of suspects or SuspectDomain
input:  $n(E)$  = number of past cases
input:  $MaxArity$ 
output: Cluedo Mystery with single answer
1 begin
2    $FinalAnswer \leftarrow$  assign variable  $X$  to store the final answer
3    $MainSuspect \leftarrow$  assign a random choice  $Suspect \in S$ 
4   for  $i \leftarrow 1$  to  $n(C)$  do
5     take the predicates involved in the rule  $C_i$  and store their arity
6     example: in  $\text{angry}(X) :- \text{hates}(X, Y) .$   $\text{angry}_{Arity} = 1, \text{hates}_{Arity} = 2$ 
7     store the location “ $l$ ” in which the Main Suspect (the final answer) may appear in each
      predicate
8   for  $i \leftarrow 1$  to  $n(P)$  do
9     if  $P_i \notin C$  then
10      randomly choose  $Arity_{P_i}$  where  $1 \leq Arity_{P_i} \leq MaxArity$ 
11      for  $k \leftarrow 1$  to  $P_{i_{Arity}}$  do
12        randomly select a Suspect from  $S$ 
13        store the suspect at location  $k$  in  $P_i$  with the Suspect
14        append  $P_i$  to output
15   for  $i \leftarrow 1$  to  $n(C)$  do
16     decompose the rule  $C_i$  into its constituent set of predicates
17     if  $C_i$  contains  $FinalAnswer(X)$  then
18       replace  $FinalAnswer$  with  $MainSuspect$  in the head and body of  $C_i$ 
19       replace the other variables in  $C_i$  with suspects  $\in SuspectDomain$ 
20       append  $C_i$  to output
21       if  $C_i = C_{\text{murderer}}$  then
22         remove the head of  $C_i$  and append the rest to output
23        $S \leftarrow S - \{MainSuspect\}$ 
24     for  $j \leftarrow 1$  to No of Predicates in  $C_i$  do
25       take  $Pred_j$ 
26       for  $k \leftarrow 1$  to  $Pred_{j_{Arity}}$  do
27         randomly select a Suspect from  $S$ 
28         replace the variable at location  $k$  in  $Pred_j$  with the Suspect
29   for  $i \leftarrow 1$  to  $n(E)$  do
30     define a new  $S$ 
31     eliminate if-condition in line 22
32     repeat from line 1 to 28
  
```

Figure 4.1: Cluedo generation algorithm

According to the algorithm shown in figure 4.1, we define a set of rules, predicates and suspects in advance. We also need to pre-define the arity of the predicates so that they should have the same arity throughout the problem. For instance, if predicate “angry” has arity 2, it should only explain the relation between two suspects or constants throughout the scenario. In line 2 we define X as “Final Answer” as it has appeared in the head of $C_{murderer}$ and will be finally replaced with a suspect that we randomly pick from the domain in line 3. Through lines 15 – 28 we replace the variables in constraints while considering that only the “*MainSuspect*” should satisfy all the constraints. We create the mystery of “who is the murderer” by eliminating the $C_{murderer}$ rule in line 22. It is also clear in line 29 that the past cases are generated similar to the current case without removing the solution.

The Cluedo generating algorithm has been implemented in SICStus Prolog. In order to program this algorithm in SICStus, we needed to build various libraries to hold: (a) predicate names along with their arity (b) rules and (c) suspects. Furthermore, it was required to make user defined functions to incorporate suspects within the predicates and rules. For instance, in order to embody the rule `murderer(X) :- loves(X,Y), tall(X)`, we had a set of functions such as *select_lover_suspects* to randomly choose suspects from their domain and incorporate them in the predicates taken from the library. In fact, such rules were formed by randomly combining the predicates in the predicate library.

4.3.1 Example of Cluedo Generation

Figure 4.2 on page 81 depicts an example of a Cluedo current case generated by GH. The past cases are given in appendix A.1.

In figure 4.2 we see the Cluedo scenario where: $MaxArity = 2$, $C = \{C_{murderer}, C_{murder_weapon}\}$,

$C_{murderer} : \text{murderer}(A) :- \text{loves}(A,B), \text{is_tall}(A).$

$C_{murderer_weapon} : \text{the_murder_weapon}(A) :- \text{was_found}(A,B),$

$\text{missing_weapon}(A), \text{potential_murder_weapon}(A).$

It is clear from the figure that apart from the murderer, a weapon also plays a supporting role

```

% at Time = 1
fact(1):- loves(white,scarlett).
fact(2):- loves(green,scarlett).
fact(3):- was_found(ropes,patio).
fact(4):- angry(green).
fact(5):- missing_weapon(dagger).
% at Time = 2
fact(6):- was_found(spanner,spa).
fact(7):- is_shortest(scarlett).
fact(8):- missing_weapon(leadpipe).
fact(9):- is_tall(plum).
fact(10):- was_found(revolver,
                    observatory).
% at Time = 3
fact(11):- is_tall(peacock).
fact(12):- missing_weapon(revolver).
fact(13):- angry(mustared).
fact(14):- is_short(green).
fact(15):- loves(mustared,scarlett).

% at Time = 4
fact(16):- is_tallest(peacock).
fact(17):- murder_scene(diningroom).
% at Time = 5
fact(18):- was_found(candlestick,hall).
fact(19):- suspect_room(diningroom).
fact(20):- is_short(scarlett).
fact(21):- is_short(mustared).
fact(22):- angry(white).
% at Time = 6
fact(23):- suspect_room(kitchen).
fact(24):- is_tall(white).
fact(25):- potential_murder_weapon(leadpipe).
fact(26):- angry(scarlett).
the_murder_weapon(A):-
    \+was_found(A,B),
    missing_weapon(A),
    potential_murder_weapon(A).

```

Figure 4.2: Example of a Cluedo scenario - current case

in the mystery. It is noteworthy that a DIP can have more than 1 suspect domain. In other words, there could be more than one mystery in a DIP. In such cases, all the DIP conditions should be held by the other domains and the same generation algorithm applies for other suspect domain(s). In this Cluedo scenario, *[scarlett, mustard, white, green, peacock, plum]* is the suspect domain for the “murderer” whereas *[candlestick, dagger, leadpipe, revolver, rope, spanner]* comprises the suspect domain for the “murder weapon”.

Similar to a detective or diagnostician, who collects clues and evidence throughout the investigation, it is clear that GH should also generate only a number of facts and/or constraints at each time step. In the future chapters where we discuss the solution methodology we will see that GH will be evaluated based on its instantaneous performance; inferences are made given the data at each time step and conclusions may thus vary with the change of information.

4.4 DIP Automatic Generation – Generic Type

In the algorithm given in figure 4.1, we specified the DIP parameters beforehand so that they ultimately describe a Cluedo scenario. For instance, we defined a set of predicates in the “user

input” section like **hate** and **love** and a set of constraints like: **hates(X,Y):-\+ richer(X,Y)**. In short, we needed to impose a meaningful relationship between predicates and suspects. However, in order to systematically generate numerous DIPs without bounding the system to produce specific scenarios, it is not efficient to handpick predicates, suspects and rules and to manually form a relationship between all of them. The only way to avoid this is to make the algorithm generic. By generic, we mean that the parameters of a DIP are defined in a way that can be instantiated and replaced by any specific term. For example, instead of predicates like **love** and **hate**, we can have a set of predicates containing **pred1** and **pred2**. This way, we will have the liberty to replace **pred1** and **pred2** by an infinite number of predicates which enables us to generate an infinite number of scenarios. The generic DIP generation algorithm has two steps: current case generation and past cases generation.

4.4.1 Generic DIP – Current Case

In the generic DIP generation algorithm, we can form the names of the suspects and predicates by just combining the acronyms and numbers without the need to specify them in the input section. For instance, by combining ‘s’ and $\{1,2,\dots,n\}$ we can form suspects’ names. However, we shall need to assign a set of parameters to construct a DIP. Figure 4.3 illustrates the required parameters.

```

number of facts: n(P)
number of suspects: n(S)
number of past cases: n(E)
number of rules: n(C)
maximum arity for each predicate: Max(arity)

```

Figure 4.3: DIP generation: current case input parameters

Unlike Cluedo, DIP generation algorithm is automatic except the first part where we need to specify a few parameters for the system; generic terms are combined randomly in order to form the names’ of predicates and suspects. The rules, the likelihood degree of suspects and the location of suspects within predicate arguments are chosen randomly. As you could see in figure 4.4 on page 84, the first for-loop combines suspect names and forms the suspect domain based

on the input parameter N_S specified in part-1. The likelihood degree of each suspect is also chosen randomly in this loop and can vary from 0 to 100 % – with 0 indicating a bystander and 100 pointing out the ultimate answer. The likelihood degree of each suspect is the proportion of the total number of rules which are satisfied by the suspect according to the likelihood definition in chapter 3 (§3.6).

The second for-loop forms the predicate names. A random number is chosen between 1 and the Max_{Arity} . This number will be assigned as the arity of each predicate. Additionally, in the third loop, some of the predicates are chosen randomly to form the rules of the DIP.

Finally, it is needed to assign the location in which the answer (main suspect) appears in the rules. We should bear in mind that main suspect(s) should hold a unique position in each rule. Let us recall from the Cluedo scenario in figure 4.2 that “`murderer(A):- loves(A,B), is_tall(A).`” indicates that murderer A , in addition to being tall, should love B . In order to embed such rules, we need to keep the occurrences of A in $C_{murderer}$ consistent. In order to do so, the final for-loop forces the algorithm to store the target locations in $C_{murderer}$ and assign the suspects to those locations. In short, suspects that appear in those locations in the rules will be considered as “role players” in the DIP scenario in contrast to “bystanders”.

Now, it is time to consider the final stage of the generic DIP algorithm in which a DIP scenario is generated. By “scenario” we mean the set of predicates and rules that represent the problem and express the relationship between suspects. We need to incorporate the suspects within the body of the rules and predicates’ arguments. As mentioned in algorithm 4.2, it is important to make sure that only the main suspect(s) – the one with likelihood degree of 100% – appears in all the rules at the target locations and the rest of the suspects should be embedded in rules’ arguments according to their likelihood degrees.

As we could see in algorithm shown in figure 4.5, *Embed_Scenario* is a function which is aimed to assign suspects in every predicate and constraint arguments. Therefore, it is required to feed essential information to the function such as constraint domain, suspect domain, the location in which main suspects should appear in the corresponding constraint and the likelihood degree of each suspect. In this part, through lines 4 – 11, we check if the likelihood degree of suspects

Algorithm 4.2: DIP Generation: current case – part 1

```

output:  $Rules(C), Fatcs(P), Suspects(S), C_{Locations}, S'$ 
1 begin
2   for  $i \leftarrow 1$  to  $n(S)$  do
3      $S_i \leftarrow String(s) + String(i)$ 
4      $AppendToSuspects(S_i)$ 
5      $N \leftarrow 0 \leq RandomNumber \leq n(C)$ 
6     calculate likelihood degree of each suspect  $S'_i$ 
7   for  $i \leftarrow 1$  to  $n(P)$  do
8      $P_i \leftarrow String(pred) + String(i)$ 
9      $AppendToFatcs(P_i)$ 
10     $P_{iArity} \leftarrow 1 \leq RandomNumber \leq MaxArity$ 
11  for  $i \leftarrow 1$  to  $n(C)$  do
12     $j \leftarrow 1 \leq RandomNumber \leq n(P)$ 
13     $C_j \leftarrow P_j$ 
14     $AppendToRules(C_j)$ 
15     $P \leftarrow P - P_j$ 
16     $n(P) \leftarrow n(P) - 1$ 
17  while  $Rules \neq \emptyset$  do
18    pick a rule  $C_k$  from the  $Rules$ 
19     $C_{klocation} \leftarrow 1 \leq RandomNumber \leq C_{kArity}$ 
20     $C \leftarrow C - \{C_k\}$ 
21     $AppendToC_{Locations}(C_{klocation})$ 

```

Figure 4.4: DIP generation: current case – part 1

is greater than 0. In order to find the number of rules, the suspect S_i satisfies, we need to multiply the likelihood degree S'_i to $n(C)$. In the cases where this number was greater than 0, we randomly pick the same number of constraints and store those constraints in S_{iRules} . For instance, if $n(C) \times S'_i = 3$, we pick 3 constraints randomly and store them in S_{iRules} .

Furthermore, through lines 14 to 26, each suspect S_i is embodied in the constraints that have been stored in S_{iRules} . Bear in mind that $C_{Locations}$ specify the main suspect locations in each of the predicates embedded in the rules. Through lines 19 – 25, we clone C_k if its arity is more than one. This means that other than S_i , we will incorporate more suspects in the constraint which results in more complicated scenario due to extra information. The purpose of checking the arity to be more than one is to avoid duplicate information; if the arity is one, there would be no empty argument to be filled with a different subset of suspects. Shuffling the suspect domain will result in a different subset of S embedded in every rule.

From line 27 to 36, we randomly incorporate suspects within the predicates that never appear

Algorithm 4.3: DIP Generation: Current Case – part 2

```

output: DIP Scenario
  /* function to embed the constraints/predicates into the background knowledge */
1 Embed_Scenario(Rules(C), Suspects(S), Facts(P), C_Locations, S')
2 begin
3
4   for  $i \leftarrow 1$  to  $n(S)$  do
5     if  $S'_i \neq 0$  then
6       for  $j \leftarrow 1$  to  $n(C) \times S'_i$  do
7          $rand \leftarrow 1 \leq RandomNumber \leq n(C)$ 
8         pick  $C_{rand}$  from  $C$ 
9         store  $C_{rand}$  in  $\langle S'_i, C_{rand} \rangle$  which is same as  $S_{irules}$ 
10         $C \leftarrow C - \{C_{rand}\}$ 
11         $n(C) \leftarrow n(C) - 1$ 
12      else
13         $S_{irules} \leftarrow \emptyset$ 
14
15    while  $C \neq \emptyset$  do
16      Pick a predicate  $C_k$  from  $C$ 
17      for  $i \leftarrow 1$  to  $n(S)$  do
18        if  $C_k \in S_{iRules}$  then
19          assign  $S_i$  to  $C_{k_{location}}$ 
20          if  $C_{kArity} > 1$  then
21             $rand \leftarrow 1 \leq RandomNumber \leq 5$ 
22            repeat
23              shuffle  $S$ 
24               $Assign(S, C_k, C_{kArity})$ 
25               $rand \leftarrow rand - 1$ 
26            until  $rand = 0$ 
27
28      Append  $C_i$  with its arguments to output
29
30     $Remain \leftarrow P - C$ 
31    while  $Remain \neq \emptyset$  do
32      pick a predicate  $P_k$ 
33       $rand \leftarrow 1 \leq RandomNumber \leq 5$ 
34      repeat
35        shuffle  $S$ 
36         $Assign(S, P_k, P_{kArity})$ 
37         $rand \leftarrow rand - 1$ 
38      until  $rand = 0$ 
39      append  $P_k$  with its arguments to output

```

Figure 4.5: DIP generation: current case – part 2

in the constraints. We also set the occurrence of each predicate to vary between 1 and 5 so that the generated scenario will be lengthier and more complicated due to addition of information.

There is a function named *Assign* in line 23. The purpose of this function is to add suspects to the rest of the argument locations of a constraint. Ultimately, each constraint is added to the output stream in line 26. This function plays an important role in mapping the suspects in the current case to the corresponding suspects in the past cases. We discussed the mapping function in chapter 3 (§3.4.3). Let us consider an example: imagine three predicates in a scenario, **pred1**, **pred2** and **pred4** where **pred1** and **pred2** appear in the rules and **pred4** does not play an important role in the scenario. A part of generated background information where the likelihood degree of x is 100% is as follows:

pred1(x) . pred2(x,_) . pred4(x,_) .

It is clear that other than **pred1** and **pred2**, x is also incorporated within **pred4**. This can happen because according to lines 27 – 36, we use the same suspect domain in order to embed suspects in the remaining predicates that do not appear in the constraints. Let us suppose that the aim is to generate a scenario in which Prof. Plum is the main suspect, commits the crime and Reverend Green is his accomplice with the same motivation. Both are tall and hate Scarlett. Therefore, **pred1(x)** is instantiated as **tall(plum)**, **tall(green)** and **pred2(x,_)** as **hates(plum, scarlett)**, **hates(green, scarlett)**. Prof. Plum also happens to be shorter than Colonel Mustard, therefore, **pred4** is instantiated as **shorter(plum, mustard)**. Being short is not an important motivation in this scenario but is a true piece of information.

Recalling the mapping function in chapter 3, figure 3.5, imagine $Y = \{y_1, \dots, y_n\}$ is a set that contains the corresponding suspects of x in past cases $\{E_1, \dots, E_n\}$. Hence, members of Y satisfy the same characteristics of x , which means that **pred4** will also be incorporated by $\{y_1, \dots, y_n\}$ in all the past cases. We will see in later in chapter 7 that this will make **pred4** an indelible part of the $C_{murderer}$ and results in Prof. Plum being categorized as the only answer with the likelihood of 100%. Since Reverend Green does not satisfy the last piece of information “**shorter(green, mustard)**”, Green will have a likelihood degree of 66% contrary to what we aimed for in the problem description.

The mapping function is an important part of the generation process and is able to change the narration of a DIP and cause different likelihood degrees for the suspects. In addition, the mapping function will produce the irrelevant information in the past cases which is akin to what an investigator may deal with in real life. A part of a DIP current case for an example is shown in figure 4.6.

```
% Likelihood Degrees
% 100 % a1
% 66 % f2
% 33 % a2
% 33 % b1
% 33 % b2
% 33 % d1
% 33 % d2
% 33 % e1
% 33 % e2
% 33 % f1
%Constraints: answer(X):- pred3(_,_ ,X,_ ,_), pred4(X), pred10(_,_ ,_,X,_).

% at Time = 1
pred1(d1,f1,b2,e2,f2).
pred1(e1,a2,d2,e2,f1).
pred2(b1).
pred2(b2).
pred3(d1,e1,a1,d2,e2).
pred3(a2,f1,d1,e1,b2).
pred3(e2,b2,b2,e1,b1).
pred3(b1,a2,f1,e1,f2).

% at Time = 2
pred3(d2,b2,b1,e1,d1).
pred3(e1,f2,d1,b2,f1).
pred4(a1).
pred4(f2).
pred4(e1).
pred5(f1).
pred5(a2).
pred5(f2).

% at Time = 3
pred6(a2,f1,b2,d2,e2).
pred6(b2,b1,f2,e1,e2).
pred7(d1,f1,b1,f2,e1).
pred8(f1,d1,a2,b2).
pred9(d1,e2).
pred9(e2,b2).
pred9(d2,b1).
pred10(d1,e1,d2,a1,e2).

% at Time = 4
pred10(d1,d2,e1,a1,e2).
pred10(b2,e2,d1,d2,b1).
pred10(d1,b1,b2,f2,f1).
pred10(d1,b1,f2,e2,d2).
pred10(e2,d1,d2,a2,f2).
```

Figure 4.6: Generic DIP – part of the current Case

4.4.2 Generic DIP – Past Cases

Generation of past cases is the last part of generation algorithm. We use the same arguments as with the *Embed_Scenario* function in algorithm 4.3. We also use the database of the *Assign* function in order to map the predicates, rules and suspects in the current case to the corresponding values in the past cases. However, the following modifications should be made to

algorithm shown in figure 4.5:

- Changing the name of the suspects according to the past case number; for instance s_{23} is the 3rd suspect in past case number 2.
- Using algorithm 4.3 from line 14 until the end. However, akin to real scenarios, we are not bound to have all the predicates and constraints in the current case to appear in the past cases. Therefore, we change the lower bound in line 20 and 30 to 0; By imposing zero as the lower bound, we randomly eliminate some of the predicates and constraints.
- With the help of likelihood degrees S' , we input the suspect(s) with 100% degree as the answer to each of the past cases. We will discuss in chapter 8 the methods we employ when there is no suspect with likelihood degree of 100%.

An example of past cases generated from the current case shown in figure 4.6 can be found in the appendix A.2

4.5 Translation Suite – TS

As mentioned in chapter §3, our aim is to build a system which generates DIPs and to tackle such problems using a combination of AI techniques. It is therefore essential to translate the DIPs into the syntaxes of the AI systems involved. To do so, we have implemented a translation suite (**TS**) that takes the problem statement as the “source language” (SL) and automatically translates it to the required syntaxes or “target language” (TL). We mentioned earlier that we decided to represent our DIP problems in first order logic, because FOL is fairly easy for expressing our knowledge; it is expressive and constrained enough which enables us to precisely describe our knowledge about the problem and we can make use of logic based tools for different reasoning techniques and ultimately Prolog [50] enables us to write programs without much difficulty.

In our project we used various programming languages and AI tools in different contexts. Recalling from chapter 3, we used Prolog, HR and OTTER apart from SICStus Prolog. In this

chapter we will also discuss the use of Weka [55]. It is worth mentioning that the translation suite, as a part of the generation module, is designed in such a way that it can translate DIPs to the required syntaxes without having any knowledge of the generation algorithm parameters. In other words, the TS performs as a stand-alone piece of software that only takes a DIP and translates it with knowing its properties such as likelihood degree of suspects, rules and facts. TS is also not aware of the nature of the arguments appearing in the predicates. This way, we can consider TS as a fully automated translation mechanism which does not get any help from the generation algorithm. The translation suite has some functionality in the solution module which will be discussed in chapter 6 (§6.3).

4.5.1 Translation to Progol

Recalling from the chapter §2, Progol is an ILP system introduced in [84] which combines inverse entailment with “general-to-specific” search. It also benefits from a refinement operator which is designed to avoid redundancy in A*- like search [123]. As an ILP system, Progol uses background knowledge and examples to learn concepts, for instance, a set of rules for predicting cancer in a patient from symptoms. By generalizing from the previous examples of cancerous and non-cancerous patients, it can learn the concept of a cancerous patient and help to diagnose the future patients. The past cases of a DIP provide appropriate examples for Progol to form a hypothesis about the solution. The only obstacle here is to massage the DIP past cases in order to fit as Progol input.

Progol syntax is almost the same as Prolog with small differences. For instance, negated examples in Prolog are expressed as `\+` whereas in Progol, negation of an example is marked by the existence of `:-` before the example. We can easily take care of this part of the translation by replacing substrings containing `\+` using Java regular expressions library. The rest of the translation is inputting the past cases of the DIP to the Progol file. However, it is essential to provide guidance for Progol in the process of generalizing examples. Without the guidance, Progol does not understand the examples appropriately and cannot learn in the most efficient way.

As discussed in chapter 2, user guidance is used to restrict the predicates that can appear in the learned rule while Progol is generalizing the examples. This is done by defining *Mode declarations*. With the help of mode declarations, we will specify which predicates can constitute the learned rule(s); in other words, we will specify the predicates that are expected to appear in the head and body of the learned rule. We want Progol to learn the rules that are embedded in a DIP scenario. However, imagine a detective has arrived at a crime scene; he/she cannot initially decide which pieces of information are important and should be incorporated into the final hypothesis about the crime and which pieces should be discarded. Using Progol constrains us to pre-define the constituents of the rule in advance and is a drawback for this system. We will discuss this in further detail in chapter 9. We need to input the past cases to Progol and define the mode declarations as part of the translation.

Specifying *modeh* declarations is straightforward. In the generic DIP, we need to search for the “*answer*” to our investigation problem. Therefore, the head of the learned rule should contain **answer(X)**. As explained in chapter 2 (§2.8.4), the *recall* of the mode declaration can be any positive number $n \geq 1$ or “*”. We choose the recall to be “*” as we are not aware of the exact number of solutions for an instantiation. The instantiation of the predicate **answer(X)** is the replacement of either input variables or output variables or constants to **X**. According to the DIP generation algorithm, the suspect who satisfies the constraint(s) will be the solution to our problem. The head of the generic constraint contains **answer(X)** and the body is a combination of predicates involved in the problem definition. Therefore, **X** in **answer(X)** is the input variable of type “*suspect*” – hence, **answer(+suspect)**. In overview, for every DIP, the translation suite must input the following in the Progol input file *modeh* declaration:

```
:- modeh(*,answer(+suspect))?
```

For the *modeb* declarations, the process is slightly different. As we are not sure about the predicates that appear in the body of the constraints, the translation suite should input all the predicates involved in the DIP as *modeb* declarations. The process of *modeb* declaration is summarized as follows:

- As shown in figure 4.4, each predicate P_i will have the same arity throughout a scenario

$(P_{i_{Arity}})$. TS will take each input string and extract the substring between the parentheses after the predicate name in order to get the arguments. Occurrence of “,” within the substring indicates the existence of more than one argument. Hence, $arity = \text{number of occurrences of “,”} + 1$.

- TS will store the arguments of each predicate and input them as the suspects in the Progol input file. For instance, upon encountering s_{21} , TS will add `suspect(s21)` to the input file.
- Since TS is not aware of the nature of the arguments involved in each predicate (whether they are suspects or constants), it will treat them as if they play a major role in the DIP scenario and will instantiate them as output variables of type “suspect”. Hence: “-suspect” should be replaced as each of the predicate arguments. The reason TS chooses the arguments to be the “-” (*output*) variable of type suspect and not “+” (*input*) will be discussed in further detail in chapter 8.
- For the same reason mentioned in *modeh*, TS will choose the *recall* as “*”.

A part of *modeb* declarations for the example shown in figure 4.6 is as follows:

```
:- modeb(*, pred1(-suspect,-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred2(-suspect))?
:- modeb(*, pred3(-suspect,-suspect,-suspect,-suspect,-suspect))?
```

As we see, *modeb* declarations are constructed based on the arity of the predicates involved in the scenario and the *recall* of each predicate is assigned as “*” due to TS not being certain about the number of alternative instantiations of a predicate.

The significant part of the translation to Progol was to automate the process of mode declarations so that Progol can have precise instructions about how to perform the generalization of examples. The rest of the translation is to simply input the past cases which are in SICStus Prolog syntax to the Progol input file. There will be also a set of parameters involved in the input file in order to control Progol execution. These parameters like `h(N)` or `posonly(ON/OFF)`

will be discussed in chapter 8. The complete translation of the past cases mentioned in appendix A.2 to Prolog is shown in appendix A.3

4.5.2 Translation to HR

Recalling from chapter 2, HR is an automated theory formation system [29] which constructs new concepts and forms conjectures using production rules and empirical conjecture making techniques. The background knowledge for HR contains a selection of concepts for a certain domain and examples of objects of interest within that particular domain. In overview, the word “concept” means the description of a set of objects of interest for HR.

The purpose of using HR is to discover potential axioms exhibited by the data which enables a solution to be found. The translation suite task is to convert the DIP past cases into the format of concepts and examples that are understandable to HR. We need to represent the concepts in HR similar to the *Objects* in Java programming language. Each concept encompasses a *definition* and a *set of examples*. The definition is a statement describing the nature of the objects of interest or the nature of sub-objects. Examples contain instances of objects or sub-objects. For every concept represented in HR, based on our need, TS will (a) provide a data table of examples (b) describe the type of objects in the data table and (c) provide a Prolog definition. The following is performed by TS on the DIP Past cases data. Note that we use the past cases mentioned in appendix A.2 for illustrating the translated segments.

- Defining the concept of “*suspect*” and taking all the suspects presented in the past cases as the examples of this concept. The lines containing the real suspects “s10” and “s11”, in figure 4.7a, are examples and the rest is concept definition.
- TS will form the `answer(X)` concept where X is a sub-object of the concept `suspect` as shown in 4.7b.
- TS will extract the arity of each predicate in the same way as mentioned in §4.5.1 and encloses each suspect with the symbol “@” as shown in figure 4.7c.

concept1	concept2	concept3
suspect(A)	answer(A)	pred1(A,B,D,E,F)
prolog:suspect(@A@)	prolog:answer(@A@)	prolog:pred1(@A@,@B@,@D@,@E@,@F@)
suspect(s10).	answer(A) -> suspect(A)	pred1(A,B,D,E,F) -> suspect(A)
suspect(s11).	answer(s18).	pred1(A,B,D,E,F) -> suspect(B)
		pred1(A,B,D,E,F) -> suspect(D)
		pred1(A,B,D,E,F) -> suspect(E)
		pred1(A,B,D,E,F) -> suspect(F)
		pred1(s10,s11,s17,s13,s19).
(a) Concept: Suspects	(b) Concept: Answer	(c) TS partial translation

Figure 4.7: TS partial translation to HR Domain File

The complete translation of the past cases mentioned in appendix A.2 to HR is shown in the appendix A.4.

4.5.3 Translation to Weka – Data Flattening for Weka

As mentioned in chapter 2, Weka [55] is a collection of powerful machine learning algorithms that can be employed for solving data mining problems. Like any machine learning tool, it aims to find new relevant information from a set of data items and hopefully use that information to execute an intelligent task. Each instance of a dataset consists of a number of attributes that are normally nominal or numerical. There are other data types which are not in the scope of this work. Weka expects the dataset to be in a comma-separated values (CSV) file format. The input dataset contains *header segment* and *data segment*. The header describes the attributes of the dataset and the data segment contains the instances in a comma-separated list. This representation is known as Attribute Relation File Format (ARFF). As mentioned earlier, GH aims to use Weka in order to extract information from the past cases and hopefully use them in order to solve the current case. Therefore, the task of translation suite is to convert DIP past cases into a Weka suitable format.

A dataset in which the instances are independent of each other can easily be turned to appropriate Weka format. In order to explain what an independent dataset is, imagine that Weka should learn about the type of weather that is suitable for playing beach Volleyball. Therefore, the beach Volleyball will be defined as “relation” in Weka header segment. We also need to

input examples of days explaining weather conditions where playing was or was not joyful. The attributes of each instance may contain temperature, humidity, wind and outlook. Assigning any value to each of the attributes will not influence the other attributes and will not force us to choose or eliminate some of the values for other attributes. An example of a Weka input file is shown in figure 4.5.3.

```
@relation beach_Volleyball
@attribute outlook {sunny, rainy}
@attribute windy {TRUE, FALSE}
@attribute temperature real;
@attribute humidity real;
@attribute playVolleyball {yes, no}
@data
sunny, FALSE, 85, 85, no
sunny, TRUE, 85, 95, no
rainy, FALSE, 60, 90, yes
rainy, FALSE, 70, 85, yes
```

Figure 4.8: Example of Weka input

Attributes like outlook can hold values like sunny and rainy while temperature can only hold a numerical value of type real. The data section after the “@” data symbol are the instances of days in which playing Volleyball is either possible (*yes* at the end of the line) or not possible (*no* at the end). The attribute values of each instance should appear in the same order as they were declared in the header section. This means that the *nth* @attribute declaration is the *nth* field of the attribute.

Adjusting the background knowledge of a DIP to fit the Weka format is difficult. The background knowledge of a DIP is interleaved, i.e., each piece of information can be connected to other pieces of information due to having common arguments. As mentioned in chapter 3 (§3.4), arguments can be *l-connected*. Therefore, the TS should restructure the DIP into the Weka format in a way that the relationships are not lost in translation.

Instead of a generic DIP, for the time being, let us use a simple medical mystery as an example; in this way we can figure out the inconspicuous obstacle of translating data to a Weka understandable format. We name this problem as *Polio* as the disease Polio is the final answer of our medical dilemma.

```

fever(infection, pneumonia).
rash(pneumonia).
rash(polio).
chronic_bone_pain(polio).
high_blood_pressure(polio).
correct_diagnosis(polio).

```

We want to employ Weka to learn about the “correct_diagnosis” given the background knowledge. We can assume the head of each predicate as an attribute. Subsequently, each attribute can hold values according to the number of arguments. For instance, fever has arity 2, so it can accommodate suspects in its first or second argument. Indeed, a suspect may appear in both or none of the argument locations. Therefore, we can present *fever* within the relation section of the ARFF file as follows:

```
@attribute fever {0,1,2,12}
```

By “0” we mean that a suspect does not appear in the predicate fever; “1” indicates that the suspect appears in the first argument and “2” means it occurs in the second argument. In addition, “12” is used when a suspect appears in both first and second arguments. Now the main problem is to present suspects.

Every suspect in the background knowledge can form a matrix where one dimension contains the attributes (in this case predicate heads) and the other dimension represents the possible locations (or the combination of locations) in which a suspect can appear. This can be programmed by using 2 interleaving hash tables in Java or C.

Let us consider polio in the above example. We assign the values according to the locations in which polio appears in each predicate. Note that we consider the predicates in the same order they have been defined in the relation section:

- Polio never occurs in the first three predicates. Therefore, the data line representing Polio starts with three “0” corresponding to the first three attributes.
- In the last four predicates, Polio appears in the first argument. Hence, four “1”s will appear in the data line after the 0’s.

- As we discussed, Weka needs to learn about the correct diagnosis. The background knowledge tells us whether a suspect appears in the “correct diagnosis” or not. Therefore, “correct_diagnosis” values are “yes” or “no”. Given that Polio is the answer, “yes” will be added as the final attribute of the suspect “Polio” in the data segment.

Figure 4.9 shows the TS output to Weka format, including the header segment and the data segment representing the suspect Polio.

```
@ relation correct_diagnosis
@attribute fever    {0,1,2,12}.
@attribute rash     {0,1}.
@attribute chronic_bone_pain {0,1}.
@attribute high_blood_pressure {0,1}.
@attribute correct_diagnosis {yes, no}
@ data
0,1,1,1,yes
```

Figure 4.9: Weka format – data segment representing Polio

There is a known problem in Weka regarding the datasets that have string attributes; like our own datasets. String and nominal data values are stored as numbers. In Weka, numbers will form the indices of an array of possible attribute values. However, the first string value is assigned index 0. This means that internally, the value is stored as 0. String attributes are normally treated as Sparse-Instances. Such instances are sparse vectors and require storage only for non-zero values. It is very likely that a string with an internal value zero will not be output. Therefore, in order to get around this problem we need to either remove zeros for the attribute indices or replace zeros with ‘?’ similar to missing values in the data segment. We can indeed remove zeros from the indices and ignore the predicate in which a suspect is absent. However, this will cause data lines to have different lengths and makes Weka prompt with an error. Instead, we replace zeros with “?” which is a standard Weka usage. Similar to Polio problem, we can translate DIP past cases to Weka based on the algorithm shown in figure 4.10 on page 97.

In the appendix A.5, we present the complete translation of the past cases showed in appendix A.2. Figure 4.11 on page 98 depicts the header segment of a DIP containing some of the

Algorithm 4.4: Flattening DIP past cases for Weka

```

input: DIP Past Cases
output: Weka CSV Input File
1 Data_Flattening(PastCase(E), Lines(L))
2 begin
3   append to output: @relation answer
4   while L ≠ 'null' do
5     li ← pick a line as String
6     if li contains substring "answer" then
7       decompose li into predicate head and argument
8       store the argument as Sanswer
9       append to output: @attribute answer {yes, no}
10    else
11      decompose li into PredNamei and arguments
12      store PredNamei in a set PredNames
13      name ← PredNamei
14      arity ← PredNamei.arity
15      append to output: @name {possible combinations like 0,1,2,12 for arity 2}
16      for j ← 1 to arity do
17        Sj ← take the argumentj
18        AppendToSuspectDomain(Sj)
19        Store j in SjPredNamei
20        if Sj appears in other locations in PredNamei then
21          store them in SjPredNamei
22        for k ← 1 to LastLine do
23          decompose the line into PredNamek and arguments
24          if PredNamek ∈ PredNames then
25            if linek contains Sj then
26              append the location of Sj in SjPredNamek
27            else
28              if linek contains Sj then
29                append to output: @name {possible combinations}
30                store the location of Sj in SjPredNamek
31      for i ← 1 to SizeOf(SuspectDomain) do
32        for j ← 1 to SizeOf(PredNames) do
33          if PredNamej ∈ SiPredNames then
34            output the locations Sj appeared in PredNamej
35          else
36            output "?"
37      if Sj == Sanswer then
38        Last Element of line = "yes"
39      else
40        Last Element of line = "no"

```

Figure 4.10: Flattening DIP past cases for Weka

predicates as attributes and the data segment containing only a single past case. The first line of data segment is the representation of “s18”. As we could see in the past cases generated in A.2, s18 appears in `answer(s18)`. Therefore, the data line finishes with “yes”. s18 also appears in the 3rd, 1st and the 4th positions of corresponding predicates 3, 4 and 10.

```
@relation murderer
@attribute pred1 {1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5, 15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345}
@attribute pred2 {1}
@attribute pred3 {1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5, 15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345}
@attribute pred4 {1}
@attribute murderer {yes, no}
@data
?,?,3,1,?,?,?,?,?,4,yes
1,1,15,?,?,15,24,4,2,124,no
2,?,1345,?,?,14,23,1,?,25,no
35,1,25,1,?,3,14,?,?,2,no
14,?,12,?,?,12,?,3,?,1234,no
45,?,245,1,?,5,3,3,2,4,no
2,1,14,?,?,?,5,14,1,134,no
3,1,235,?,?,24,14,?,?,135,no
?,1,134,?,1,24,25,2,?,25,no
?,?,35,?,1,35,13,2,1,135,no
```

Figure 4.11: Part of a DIP translated to Weka

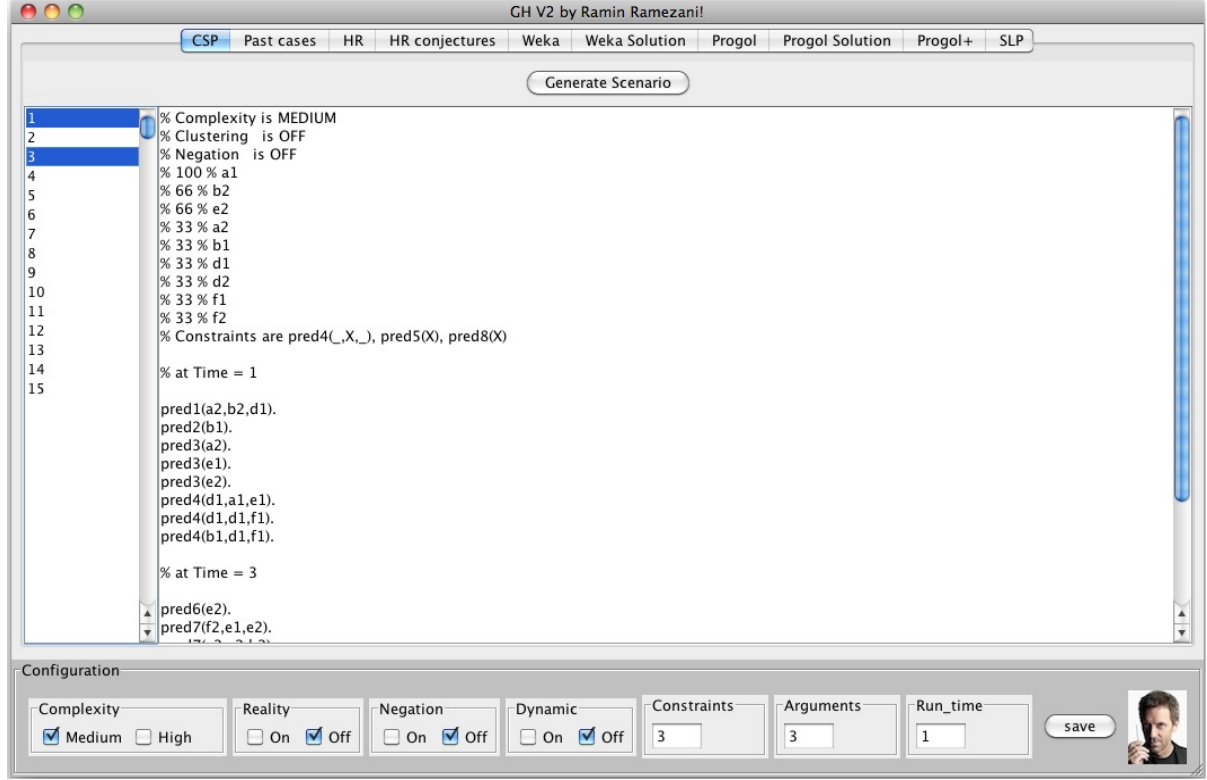
4.6 Conclusions

In this chapter we introduced algorithms that can systematically generate dynamic investigation problems. Initially, we considered a board game known as Cluedo due to its similar characteristics to DIPs. We developed an algorithm that can automatically generate Cluedo scenarios. We further realized that in order to systematically generate numerous DIPs, it is not practical to handpick predicates, suspects and constraints and to manually form a relation between all the parameters. Therefore, we modified the generation algorithm to be generic. The parameters of a DIP are defined in such a way that can be instantiated by any specific term. Algorithms shown in figures 4.4 and 4.5, explained DIP generation in detail.

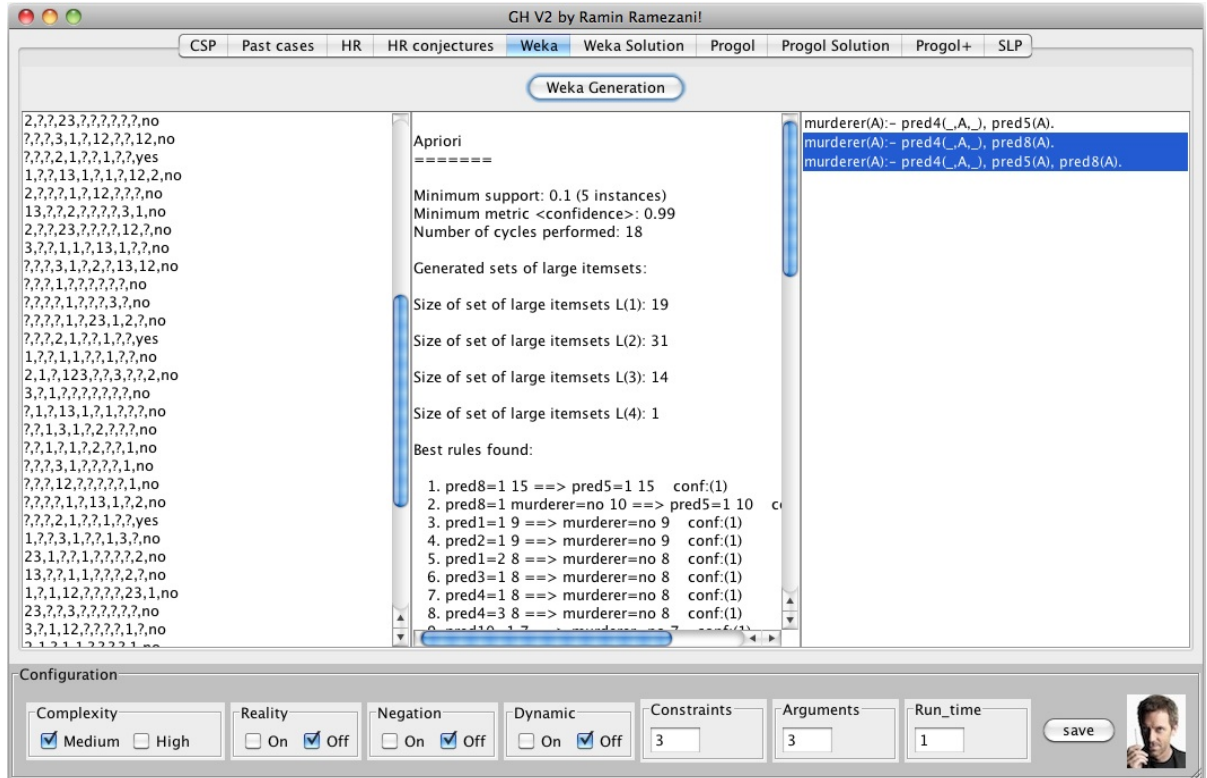
Through generation, we created scenarios with various levels of difficulty that can finally gauge the efficiency of the solution module. We mentioned earlier that we aim to employ different reasoning techniques in order to tackle DIPs; we ultimately need to input the DIPs into various AI systems. Hence, the need of an interlingua or an automated translator is evident. The translator automatically takes a DIP and modifies its syntaxes in order to be understandable by different AI systems. We explained the techniques for translating DIP into Progol, HR and Weka syntax. As mentioned in 4.5.3, DIP data, unlike the attribute-value pair machine learning problems employing Weka, are interconnected. Every argument in a predicate may appear in other predicates. This type of data is unconventional in Weka standards. We introduced a way to change the structure of a DIP and reformat it to Weka syntax.

The system we aim to introduce throughout this thesis is supposed to automatically generate DIPs, translate them and finally solve them using various AI techniques. We named our system “GH” after the protagonist of *House, M.D.* television series, Gregory House. We mentioned in §3.2 that stories of Sherlock Holmes and House, M.D were both inspirational to our work and since only one name would suffice, we picked GH. Figure 4.12 on page 100 depicts the GUI of the GH system. It shows two of the GUI screens related to current case scenario and Weka generation. As shown in the figure, every module involved in GH is associated with a button that calls and executes the relevant function upon pressing.

Constituents of GH System such as generating current case, past case, solution and so on, are shown in figure 4.12. The first block “CSP” accommodates the current case and the button “*generate scenario*” is to generate a CSP. On the left side of the GH in the CSP panel, we see a set of numbers that indicate the time steps. Each of the numbers in blue imply the data obtained in that particular time step and consequently, if all the numbers are activated, the entire scenario is presented. The other panels, HR, Weka and Progol, hold the corresponding data. “*Progol +*” will be explained in the solution chapter 7. There is a “configuration” panel on the bottom of the GUI containing some parameters. This panel enables us to modify the difficulty of a DIP generation algorithm and will be discussed in further detail in chapter 8.



(a) current case scenario



(b) Weka generation

Figure 4.12: The GH System GUI

Chapter 5

GH - The Reality Factors

5.1 Motivations

In chapter 4, we introduced our system, GH, that has been named after a fictional medical investigator, Gregory House. One of GH's tasks is to generate a type of hybrid AI problem called Dynamic Investigation Problem (DIP) that we formally defined in chapter 3. A DIP is designed to model real life investigation scenarios in which a guilty suspect has to be chosen from a number of suspects. A DIP contains a set of suspects that can be presented as suspected criminals or suspected illnesses, etc. It also contains facts about the current case and many other cases that bear resemblance to the current one. Similar to real life scenarios, the information keeps changing at different times - hence the dynamic nature of the problem. The GH system is supposed to ultimately identify the guilty party or at least reduce the number of suspects by ruling out the bystanders. Initially, as shown in §4.3, GH generated Cluedo scenarios. Later, GH was designed to form problems that are more complicated than Cluedo. In chapter 4, we described the methods of generating DIPs. Recalling from §4.4.2, the algorithm for generating past cases was slightly different to the algorithm responsible for generating the current case. The main difference was the random elimination of some of the predicates representing the background knowledge. This elimination was to make the past cases more realistic as in the real life scenarios, past cases are not always identical to the current case. However, as mentioned in

§3.2, part of this research’s goal is to design DIPs that represent real life investigation scenarios. In order to do so, we need to design past cases to have more dissimilarity to the current case. In this chapter, we will discuss the factors that we call “*Reality Factors*” and will allow GH to generate more realistic scenarios. We will also describe how we tuned GH in order to apply such changes to algorithm for generating past cases.

5.2 Reality Factor: Weighted Predicates and Suspects

We looked at how to generate investigation problems in the early stages of the development of GH. These problems are known to GH as current cases as GH assumes each investigation problem as a current dilemma, and aims to solve them by employing different AI techniques. As we explained in chapter 3, by generating the past cases, we aim to employ various techniques in order to learn the profile of the guilty party. This profile will be applied to the current case in order to solve the problem or to make the problem easier by highlighting the bystanders or eliminating some of the suspects.

Recalling from chapter 4 (§4.4.2), GH uses the same generation algorithm with slight modifications for producing past cases. Modifications were made to avoid generating past cases identical to the current case. Let us consider a medical situation that we call the *Siberia* problem: *A patient is admitted to a hospital upon returning from Siberia. His symptoms include fever, rash, chronic bone pain and high blood pressure. The doctor knows that it is very common to contract **Pneumonia** in cold weather and pneumonia explains two of the symptoms. However, another disease called **Polio** explains three of the symptoms. It is prudent to examine both of the diseases, but the patient’s condition is deteriorating and the doctor cannot afford the time required by different tests. A wise doctor should always prioritize his/her judgment.*

We present the *Siberia* problem according to the DIP formal definition described in chapter 3 (§3.5) and we show it in figure 5.1. As explained in chapter 4, the GH generation engine can produce numerous similar problems by replacing the symptoms and suspects with generic predicates and arguments say `pred1(a,b,c)` or `pred2(d)`.

```

set of suspects    S = {cold_weather, pneumonia, polio }

symptom(fever)
symptom(rash).
symptom(high_blood_pressure)
symptom(chronic_bone_pain).
cause_of(fever, cold_weather).
cause_of(fever, pneumonia).
cause_of(rash, pneumonia).
cause_of(rash, polio).
cause_of(chronic_bone_pain, polio).
cause_of(high_blood_pressure, polio).

```

Figure 5.1: Siberia problem in GH format

While analyzing the above example and the methodology for generating past cases, a few critical notes came to light that we named “*reality issues*” as follows:

- 1) GH should generate past cases that are not very similar to the current case, yet hold common attributes to it. This is due to the fact that in real world scenarios, past cases are normally not identical to the current case and only hold some resemblance to it. Nonetheless, identical past and current cases may also occur which makes them easier to solve.
- 2) Not all the symptoms are equally important. In overview, some of the facts, suspects and rules in a DIP are less significant than others. GH should also find a way to deal with problems in which some important information might be missing.
- 3) In real life scenarios, some information may be misleading. For instance, a patient may have a symptom which is irrelevant to his disease and the problem can be solved by eliminating that piece of information.
- 4) GH needs to find a way to check whether there is an analogy between a current case and the past cases.

note on 1:

Past cases that are very similar to the current case may imply that GH is generating problems that even though they may seem complicated, can be solved with the help of machine learning tools. It raises the concern of generating past cases with dissimilarities with the current case. We

have overcome this problem by finding a measure of similarity between the cases and applying it to GH generating algorithm. This way we can dictate GH to design past cases with a certain degree of dissimilarity.

note on 2:

Another concern is that not all the symptoms are equally important. In the example shown in figure 5.1, given that the patient's condition is deteriorating and timing is crucial, the concern is to decide which of the suspects (Polio or Pneumonia) should be examined first. We will explain in chapter 7 that GH aims to find all possible conjectures from the past cases and will eventually rank the answers based on the current case background information. However, in this particular case, GH suggests that Polio is more likely to be the correct answer as it fits more of the symptoms and pneumonia is the second best diagnosis. Let us write down this situation according to the formal definition of a DIP described in chapter 3 (§3.5):

$$\begin{aligned}
 \text{Let: } & P = \{\text{set of facts}\}, \quad S = \{\text{set of suspects}\} \\
 & \exists s_1 \in S : s_1 \text{ appears in } P_M \text{ such that } P_M \subset P \\
 & \exists s_2 \in S : s_2 \text{ appears in } P_N \text{ such that } P_N \subset P \\
 & \text{sizeof}(P_M) < \text{sizeof}(P_N)
 \end{aligned}$$

We can replace s_1 by Pneumonia and s_2 by Polio which indicates that Polio satisfies more symptoms than Pneumonia. However, such scenarios can be disguised during the process of generating past cases; as discussed in chapter 4 (§4.4.2), GH randomly incorporates suspects in the predicates and according to algorithm 4.3 (line 30), number of predicates also varies. In other words, " $m < n$ " in the above equation can change, so do the number of predicates each suspect satisfies. In this particular example, polio satisfies 3 of the symptoms and pneumonia only 2. In case the number of symptoms that Polio satisfies in the past cases becomes less than Pneumonia, the importance of the suspect polio is altered.

note on 3:

It is possible that no single disease (answer) can explain all the symptoms. This means that

we cannot build a theory that explains all the symptoms. There is always a possibility that a symptom is just a coincidence or caused by a separate benign condition. In such scenarios, a detective or a diagnostician may infer the answer by eliminating one or more of the symptoms. Let us write down this situation while recalling the formal definitions explained in chapter 3.

definitions 3.4 and 3.5: C is the set of rules and P is a set of facts.

definition 3.8: A_i and A_j consist of ground unit clauses of a single target predicate **answer** where the ground values are the suspects.

definition 3.6: S'_i and S'_j contain likelihood degrees of suspects in A_i and A_j respectively.

$$P \wedge C \wedge S \models A_i \quad (5.1)$$

$$\exists P_k \in P \text{ such that } P - \{P_k\} \wedge C \wedge S \models A_j \quad (5.2)$$

$$\text{comparing } A_i \text{ and } A_j : S'_j > S'_i \quad (5.3)$$

As shown in the above formulae, the elimination of the P_k from P in 5.2, has resulted in an answer with higher likelihood degrees.

note on 4:

Imagine in a medical investigation problem that a doctor tries to find analogies between a current case and many cases he/she has encountered through years of practicing medicine. Remembering the analogous cases may enable the doctor to test and apply the information from those cases to the current case and hopefully find the profile of the guilty suspects easier. Similar to the note 1, the doctor needs to measure the resemblance between the current and past cases. The process of solving problems based on the solution of the similar past cases is called “case-based reasoning” [73]. The main obstacle is to find a way to implement a *retrieval* stage which is to extract the similar cases from a set of past cases.

5.2.1 Dealing with Reality Issues

Dealing with problems that contain irrelevant information, as discussed in note 3, requires more explanation that will come in chapter 7 (§7.4). In this section, we explicate methods by which

we try to overcome the concerns mentioned as “reality issues”.

Tackling issue 3:

We will show in the chapter 7 (§7.4) how GH will employ *Pe-Pl* [28, 27, 40], which is a parameter estimation software in Prolog, to learn the probabilistic model of the current case. GH will be able to calculate the probability of the set of answers A according to the set of predicates and suspects. With the use of *Pe-Pl*, GH can choose the answer with higher likelihood degrees of the suspects (S'). This means that predicates that result in a lower likelihood degrees of answers will be eliminated from the set of predicates in the background knowledge. This process will be described in more detail in the solution chapter.

Tackling issue 4:

Issue number 4 raises the topic of case-based reasoning (CBR) [65]. One of the main steps in the CBR process is called *retrieval* that is basically a mechanism that recalls the similar useful cases at proper timings. A CBR system uses different steps such as *reuse* and *revise* in order to map the previous cases' solution to the current case and possibly revise the former solutions to fit better in the current case. However, the main issue which raised the question number 4 was how to find analogies between the past cases and the current case – hence retrieval. According to chapter 3, we expect DIPs to already contain similar past cases. In fact, the artificial generation of past cases in chapter 4 and the current chapter is to provide DIPs with past cases that are useful for solving the current case and we will finally show how to solve these problems. The retrieval stage of case-based reasoning is not in the scope of this thesis, yet, can motivate us in future version of GH that we will explain in chapter 10 (§10.2).

Tackling issues 1 and 2:

A simple way to avoid the identical generation of cases which raised the first issue, is to alter the number of predicates appearing in the past cases. This will result in another problem mentioned in note 2; corresponding suspects may satisfy different number of symptoms in the past cases. Based on the past cases, we may not be able to correctly decide which suspect has the higher likelihood degree to fit the answer. Therefore, we realized the need to classify the information in the current case into groups of higher and lower important information.

Systems such as Protégé developed by Stanford University [22], and also CYC, developed by a company called Cycorp [1], are both ontology editors that aim to develop a comprehensive data base from common sense knowledge. Protégé has a tool for medical data analysis which is basically a huge database of real medical histories. Upon entering few symptoms, it uses data mining techniques to extract information about similar patients so that it can advise on the current patient. CYC, on the other hand, given a meaningful word, tries to output similar words like a dictionary.

We aimed to use these systems to provide us with the additional words that are likely to share the same concept of a given word rather than its synonyms. For instance, given the word “murder”, we were expecting to extract “*weapon*”, “*hatred*” or “*poverty*” from the system as these words are likely to appear in a *murder* context based on our general notion of a murder. In that case, GH could have entered predicate names into the CYC or Protégé and consequently rank the importance of the predicates autonomously in regard to the other predicate names. In case of a murder scenario, predicates with hatred and financial issues would have obtained higher importance than predicates indicating boredom of a suspect. CYC, similar to a dictionary, could only output synonyms and was not able to infer the context from a given set of words and Protégé is only recently being considered to be improved for biomedicine. However, this remains a challenge for further development of GH.

We tried to tackle the issues 1 and 2 by developing a heuristic way to deal with the background information of the current case. GH extracts the important information in the current case by categorizing the predicates and suspects into two groups: “*higher importance*” and “*lower importance*”. With the help of this categorization, GH generates the past cases; in other words, GH, allocates information in the current case into two groups and uses the same groups while generating past cases. First, let us formalize this situation in the following formulae:

$$\begin{aligned}
 &\text{Let: } P = \{p_1, \dots, p_n\} \quad \text{be the set of facts} \\
 &\exists P_{High} \subseteq P, \exists P_{Low} \subseteq P \quad \text{such that} \\
 &\quad P_{High} \cup P_{Low} = P \quad \text{and} \quad P_{High} \cap P_{Low} = \emptyset
 \end{aligned} \tag{5.4}$$

$$\begin{aligned}
&\text{Let: } S = \{s_1, \dots, s_n\} \text{ be the set of suspects} \\
&\exists S_{High} \subseteq S, \exists S_{Low} \subseteq S \text{ such that} \\
&\quad S_{High} \cup S_{Low} = S \text{ and } S_{High} \cap S_{Low} = \emptyset
\end{aligned} \tag{5.5}$$

P_{High} and P_{Low} in formula 5.4 are “higher importance” and “lower importance” subsets of predicates respectively. The same applies to S_{High} and S_{Low} in 5.5. The above formulae imply that the intersection of high and low important groups is an empty set. In other words, once a piece of information is allocated to a group, it cannot be presented in the other group.

This way: (a) GH uses a heuristic function (estimation) to categorize the information in the current case; therefore, the past cases will not be identical to the current case (b) depending on the estimation, GH keeps the same pattern of information for the past cases and (c) this estimation is the first attempt of GH finding the solution without using any particular AI technique such as machine learning (we will discuss this in further detail in chapter 7 (§7.4)).

The main issue here is how to define **importance**. Importance of a piece of information is the level of priority or hierarchy it holds in order to be considered, comparing to the other pieces of information. The heuristic strategy is to *prioritize* the current case information; GH assigns **weight** to the information in the current case. At the same time, GH imposes a level of similarity to the past cases by specifying which part of the data belonging to the current case has higher weight so that it will generate past cases with respect to that.

In a constraint satisfaction problem solving, weighting the constraints has long been a concern. There are various approaches for weighting the constraints. For instance, in *Weighted CSPs* (WCSP), as discussed in chapter 2 (§2.5.1), a set of cost functions are declared in order to set a preference among various solutions [114]. In other words, a weight is assigned upon violating a constraint based on predefined preferences. There are other approaches including *Dynamic Flexible CSP* (DFCSP) that overcome the weakness of classical CSPs and static structure of CSPs by using flexible and dynamic constraints [81]. In all these methodologies, some sort of weighting is applied to the constraints. However, our problem is to build a model from the current case in order to generate past cases based on the concerns mentioned in the issues

1 and 2. Currently, we are not describing the solution methodology yet and the purpose of weighting background information which includes suspects and predicates is to have a more specific current scenario rather than solving it. We enabled GH to implement *weighting* by ranking the information in the current case based on a certain criteria. We will describe what we mean by *criteria*, shortly.

We should bear in mind that DIP problems are generated randomly and we already justified random generation in the generic section of chapter 4 (§4.4). Due to random generation, DIPs can exhibit random or stochastic behavior. This means that even though GH tries to categorize the information in the current case, this method is non-deterministic. In other words, weighting the information will yield an approximate categorization of higher and lower important information.

Prioritizing the information can be done in two different ways:

- a) GH end-user (similar to a detective or a doctor) can prioritize information manually, say, in the Siberia problem mentioned in 5.1, a doctor initially suggests that **rash**, **fever** and **polio** are more important. GH currently has an option that enables the user to specify the important predicates beforehand. This is part of the GH's graphical user interface objective to be ultimately used as a prototype system aiding doctors and/or detectives. We will describe this in more detail in chapter 10 (§10.2).
- b) The other criteria for prioritizing the information (predicates and suspects) is the frequency of occurrence. The frequency of occurrence of a symptom or a suspect may indicate the level of importance. As explained in chapter 3 (§3.5), dynamic nature of a DIP guarantees the flow of information at different timings for a limited time. Imagine a detective is struck by various information about a suspect every step he takes to solve the mystery or imagine a doctor that still finds high blood pressure in a patient despite different treatments he performs during the diagnosis. Frequency of information in a DIP is a criteria than can indicate the importance and GH uses this criteria in order to weight the information in a current case.

By weighting the information, GH is equipped with a more realistic generation algorithm. Past cases will be generated based on the importance of each predicate and suspect in the current case. This way, the dynamic aspect of a DIPs is also considered more realistically while generating past cases. Imagine at time T only part of the background knowledge is revealed and hence the past cases are only based on that particular segment of information. Based on P_{High} , P_{Low} , S_{High} and S_{Low} defined in formulae 5.4 and 5.5, we formalize the prioritizing process as follows:

$$\begin{aligned}
&\text{At time } T = 1 \quad \text{let: } P_1 = \{p_1, \dots, p_m\} \quad \text{and} \quad S_1 = \{s_1, \dots, s_m\} \\
&\exists P_{1High} \subseteq P_1, \quad \exists P_{1Low} \subseteq P_1 \quad \text{such that} \\
&\quad P_{1High} \cup P_{1Low} = P \quad \text{and} \quad P_{1High} \cap P_{1Low} = \emptyset \\
&\exists S_{1High} \subseteq S_1, \quad \exists S_{1Low} \subseteq S_1 \quad \text{such that} \\
&\quad S_{1High} \cup S_{1Low} = S \quad \text{and} \quad S_{1High} \cap S_{1Low} = \emptyset \tag{5.6} \\
&\quad \vdots
\end{aligned}$$

$$\begin{aligned}
&\text{At time } T = n \quad \text{let: } P_n = \{p_1, \dots, p_n\} \quad \text{and} \quad S_n = \{s_1, \dots, s_n\} \\
&\text{formula 5.6 holds between } P_{nHigh}, P_{nLow}, S_{nHigh}, S_{nLow} \\
&P_{nHigh} \cap P_{1High} \geq \emptyset \quad \text{and} \quad P_{nLow} \cap P_{1Low} \geq \emptyset \quad \text{and} \\
&S_{nHigh} \cap S_{1High} \geq \emptyset \quad \text{and} \quad S_{nLow} \cap S_{1Low} \geq \emptyset \tag{5.7}
\end{aligned}$$

It is clear from 5.7 that the sets of *high* and *low* important suspects and predicates may or may not have any common element at different timings. In other words, once all the background information about the current case is achieved, the categorization of higher and lower important information may be totally changed.

Note: Recalling from chapter 3, background information of a current case DIP: <suspects, constants, predicates, rules, case studies, mappings>. In the process of weighting the background information, GH takes suspects(S) and predicates(P) into considerations and avoids dealing with rules. GH considers rules as constraints that are needed to be satisfied and therefore, should all be presented in the past cases as well. However, we will explain in the solution chapter that it is possible to find the probability of the rules with the use of *Pe-Pl* [28] and possibly solve the case with the use of dynamic and flexible CSP [81].

GH needs to perform two tasks in the current case scenario of a DIP: (a) Ranking predicates according to their occurrences and (b) Ranking suspects based on their occurrences in the high-ranked predicates. The rank of suspects and predicates is calculated according to the following formulae. We should note that for ranking the suspects, the location in which they appear in a certain predicate is also important. As explained in generation algorithm §4.3, suspects can appear in different locations of a predicate and hence, they can hold different ranks accordingly.

$$\text{rank of each predicate } P_i = R_{P_i} = \text{number of occurrences of } P_i \text{ in the problem} \quad (5.8)$$

Based on the formula 5.8 and the threshold we describe later, we assign predicates as *high importance* or *low importance*. The rank of each suspect, is not only based on their occurrences. Unlike predicates, suspects have another attribute that is the location in which they are represented in a predicate. The occurrences of a suspect x , is calculated based on the location it appears in a predicate P_k :

$$N_{x_{j,k}} = \text{Number of occurrences of } x \text{ in } P_k \text{ at location } j \quad (5.9)$$

$$\text{rank of each suspect } x = R_x = \sum_{k=1}^n (R_{P_k} \times \sum_{j=1}^{\text{arity}_{P_k}} N_{x_{j,k}}) \quad (5.10)$$

Equation 5.10 shows that rank of each suspect x is the product of the summation of the ranks of P_K (R_{P_K}) multiplied by the number of occurrences of x in P_k . According to the equation 5.10, GH counts the set of predicates that are being selected as *high importance* predicates. Therefore, the ranks of suspects are calculated if they appear in the *high importance* predicates. Note that location j is the argument position in a predicate in which a suspect may or may not appear. Bear in mind, the rank of each predicate will be multiplied in the equation, given the suspect appears in any position in the predicate, otherwise, the rank will be multiplied by “zero”.

By implementing the above formulae on the current case, GH classifies a number of suspects and predicates in the *high importance* group and the rest in *low importance* group. GH uses these categorizations to form past cases. Note that during the process of generating past cases

from the categorization of suspects and predicates, GH assigns the high ranked suspects as the solution to the past cases.

The problem here is to adjust a proper threshold on the suspect and predicate ranks. Choosing the appropriate threshold is experimental. This threshold should differentiate the high and low importance data. When the threshold is too low, the majority of the suspects and predicates will be categorized in “high importance” group. If the past cases contain almost all the predicates in the problem, then the machine learning tool may over-fit our problem and come up with a trivial hypothesis.

Similarly, if the threshold is too high, only a few suspects and predicates will be categorized in the high importance group, so that only a few suspects and predicates will be counted as important. This way, the machine learning may not learn any hypothesis at all.

Based on this categorization, GH will derive an estimation of important suspects and predicates in the problem and, therefore, past cases will not be completely analogous to the current case. The other advantage of using weighted predicates and suspects is that GH can generate past cases at each time step based on the partial information provided in the current case. This way GH generates more realistic scenarios.

5.2.2 Example of Generation Based on Weighting

We initially show an example of a current case. GH calculates the rankings of predicates and suspects and generates past cases according to the rankings. Figure 5.4 on page 114 illustrates all 4 time steps of a current case. GH initially classifies the important predicates based on their number of occurrences. As you can see, the rule contains **pred1**, **pred3** and **pred9** where the answer should appear in the 3rd, 2nd and 1st arguments respectively.

Based on the number of occurrences, GH classifies the following predicates as “*high importance*” predicates: [**pred1**, **pred2**, **pred3**, **pred9**]. It is clear that GH has also classified **pred2** in high importance predicates. Figure 5.2 shows the occurrence of suspects in the important predicates along with their position. For instance, **d1={pred3=1, pred9=1}** means that the suspect

d1 appears in 2 of the 4 important predicates (**pred3**, **pred4**) and in their 1st argument.

```
{d1={pred3=1, pred9=1}, d2={pred3=2}, a1={pred1=3, pred3=2, pred9=1},
b1={pred2=3}, a2={pred2=2, pred3=1, pred9=1}, e2={pred1=1, pred3=2},
f1={pred1=2, pred2=1, pred3=2}, b2={pred2=2, pred9=1}, f2={pred1=3,
pred2=3, pred9=1}, e1={}}
```

Figure 5.2: Occurrence of suspects in the important predicates

According to figure 5.2, the rank of suspects in descending order is shown in figure 5.3.

```
{d1=17, a1=16, a2=14, f1=14, f2=13, e2=11, b2=8, d2=6, b1=3, e1=0}
```

Figure 5.3: Ranking of suspects

As an example, we see in figure 5.4 on page 114, **d1** appears in many predicates, however, only **pred3** and **pred9** are accountable for calculating the rank as shown in table 5.2: **d1**={**pred3**=1, **pred9**=1}. **d1** occurs twice in **pred3** which has rank 6 and happens only once in **pred9** with the rank 5. Therefore, the rank of **d1** = $(2 \times 6) + (1 \times 5) = 17$. The rest of the suspects are calculated in the same way.

It is worth mentioning that **e1**, according to figure 5.2, appears in none of the important predicates and, therefore, has been assigned with the rank 0. However, it is clear in figure 5.4 that **e1** appears in **pred3** which is an important predicate. When a suspect appears in, say, two different locations in the same predicate, GH considers them as two different suspects and assigns them with different rankings. For instance, if a suspect happens in the first argument of a predicate twice and at the same time appears just once in the second argument of the same predicate, higher priority would be given to the first location of the predicate. However, if a suspect obtains the same rank for both the arguments of a predicate in which it appears, GH assumes that suspect as *trivial* and allocates 0 to its ranking. In this case, **e1** appears once in both the 1st and 2nd argument of **pred3** and the same scenario happens to **e1** in **pred2** – hence the rank 0. When the number of occurrences is more than 1 for any number of locations, then GH assigns weight to that suspect for all its positions. This is due to assigning a degree of importance to the suspect that appears more than once.

We should also note the difference between the two sets of suspect hierarchies: (a) the order

```

% Likelihood Degrees
% 100 % a1
% 66 % a2
% 66 % b2
% 66 % d1
% 66 % f2
% 33 % d2
% 33 % e1
% 33 % e2
% 33 % f1
% rule: answer(X):- pred1(_,_,X), pred3(_,X), pred9(X).

% at Time = 1

pred1(d1,d2,a1).
pred1(b1,d1,a2).
pred1(e2,f1,b2).
pred1(a2,b2,f2).
pred1(d2,b1,d1).
pred2(e1,a2,b1).
pred2(d1,e1,f2).
pred2(f1,b2,d1).

% at Time = 2

pred3(d1,a1).
pred3(d1,f1).
pred3(a2,e1).
pred3(f1,f1).
pred3(e1,e2).
pred3(f1,d2).
pred4(b1).
pred4(e2).

% at Time = 3

pred5(b2,e2,f2).
pred5(f2,b2,b1).
pred6(e2).
pred7(f1).
pred7(b1).
pred7(d2).
pred8(d2,e1).
pred9(a1).

% at Time = 4

pred9(a2).
pred9(b2).
pred9(d1).
pred9(f2).
pred10(e2,b2).

```

Figure 5.4: Part of a current case

of suspects according to their ranks (b) the order of suspects based on their likelihood degrees shown in figure 5.5.

Figure 5.5 confirms that GH's ranking system does not exactly produce the same suspect's rankings as we embedded in the current case. For instance, "a1" is the highest priority suspect which is embedded in the current case as shown in figure 5.4, whereas, GH has allocated "d1" as the first and a1 as the second highest ranked suspect with slightly less possibility. GH has also given a small weight (3) to b1, while this suspect has the likelihood degree of 0 in the current case, so that it does not appear in the list of degrees in figure 5.4.

```

{d1=17, a1=16, a2=14, f1=14, f2=13, e2=11, b2=8, d2=6, b1=3, e1=0}
{a1=100%, a2=66%, b2=66%, d1=66%, f2=66%, d2=33%, e1=33%, e2=33%, f1=33%}

```

Figure 5.5: Ordering of suspects

Based on the rankings, GH generates the past cases with the help of mapping function as we explained in chapter 4 (§4.4.2). Initially, GH allocates the *high importance* predicates in every rule. Later, it generates a set of suspects and maps them to the suspects in the current case. Therefore, the calculated rankings will be assigned to the new set of suspects. Further, GH replaces the suspects in the current case with their corresponding suspects in the newly generated set of suspects. We should note that only high ranked suspects in the *high importance* predicates are mapped to the past cases and the rest of the suspects are replaced randomly with the suspects from the new set. For instance, upon reaching **d1** in **pred3** which is an important predicate, **d1** is replaced by its corresponding suspect from the new set of suspects which is **s10**.

We should bear in mind that GH's weighting system is a stochastic way of ranking suspects, so the obtained categorization from the current case, though analyzed statistically, tries to derive an approximate set of important information. Therefore, with this methodology the important location of the predicates in which the *main suspects* appears, may also change. As we could see in figure 5.4, the *main suspect* appears in the second argument of **pred(3)**, whereas, the first argument of **pred3** is apparently the position for main suspects as we can see in the generated past cases in figure 5.6.

We will discuss in chapter 7 how stochastic generation of past cases based on rankings can also be used as an approximate solution to the current case with convincing results – in case GH does not use the solution module. We will discuss this further in chapter 7 (§7.4). However, as we can see in figure 5.6, GH past case generation based on the rankings produces scenarios where the background information in the current case have not been exactly mapped to the past case.

The other advantage of this method is the ability to produce past cases according to each time step of the current case. In this way, GH, ranks only the partial current case data and outputs the past cases based on that particular piece of information.

As shown in figure 5.7 on page 117, the past case is generated based on a single piece of information provided at time $T = 2$. GH has allocated **pred4** to *low importance* predicate and

```

% rule predicates: [pred1,pred2,pred3,pred9]
answer(s10).
answer(s11).
answer(s12).
answer(s13).
pred1(s10,s13,s11).
pred1(s15,s13,s16).
pred1(s12,s16,s17).
pred2(s18,s12,s14).
pred2(s10,s18,s17).
pred2(s13,s16,s10).
pred3(s10,s11).
pred3(s10,s13).
pred3(s12,s18).
pred3(s13,s13).
pred9(s11).
pred9(s12).
pred9(s10).

```

Figure 5.6: Example of a past case based on rankings

hence does not appear in the past case. As shown in figure 5.8 on page 117, **e1** appears once in the first and second argument of **pred3** – hence the rank 0. In addition, we can see that the suspects appearing in **pred4** are not considered in the rankings and **e2** is only ranked due to its occurrence in **pred3**.

In summary, with the stochastic way of forming the pattern of the current cases, GH avoids identical current and past cases. GH weights the information based on their importance by which more realistic past cases are generated. However, an end-user can also specify the important information for the generation module.

5.3 Reality Factor - Negation

In order to make DIPs more realistic, we need them to include negations. In the generation algorithms discussed in chapter 4, we did not mention negated predicates. However, it is essential for GH to be able to generate scenarios which include negation. For instance, we may know that a suspect is not tall in a murder scene, therefore, **not(tall(suspect))**. We modified

```

% rules:
% answer(X):- pred1(_,_ ,X),
%      pred3(_ ,X), pred9(X)

% at Time = 2                                % rule predicates: [pred3]

pred3(d1,a1).                                answer(s40).
pred3(d1,f1).                                answer(s41).
pred3(a2,e1).
pred3(f1,f1).                                pred3(s40,s41).
pred3(e1,e2).                                pred3(s40,s42).
pred3(f1,d2).                                pred3(s43,s43).
pred4(b1).                                    pred3(s42,s42).
pred4(e2).                                    pred3(s42,s45).

```

(a) Current Case at Time $T = 2$ (b) Past Case based on $T = 2$

Figure 5.7: Past case generation based on partial current case ranking

```

{d1={pred3=1}, a1={pred3=2}, f1={pred3=2}, a2={pred3=1},
e1={}, e2={pred3=2}, d2={pred3=2}, b1={}}
Rank: {d1=12, a1=6, f1=6, a2=6, e2=6, d2=6, e1=0, b1=0}
Rank{d1=17, a1=16, a2=14, f1=14, f2=13, e2=11, b2=8, d2=6, b1=3, e1=0}

```

Figure 5.8: Rank of suspects based on $T = 2$

the generation algorithm to add some extra negated background information. GH adds “not” to some of the predicates. This modification is quite straight forward. However, if any of the predicates which are supposed to appear in the rules are negated, then GH should embed the “main suspect(s)” according to the negated predicates. Therefore, the negated predicates will be the member of “*Rules*” defined in chapter 3. It is also possible that both the predicate and the negated predicate are members of *Rules*. In this case, suspect allocation in the constraints from lines 14 – 26 in algorithm 4.3 will be based on the predicates in which suspects will appear.

For our translation suite, we need to adjust Progol and Weka accordingly. A negated predicate indicates that the arguments will certainly not appear in the predicate in the given form. Therefore, in the case of Weka, the addition of the “{0}” element is essential in both the data and the header segment. However, as explained in chapter 4 (§4.5.3), while dealing with string attributes and sparse vectors the index 0 may cause problems. Similarly, Progol also has a strange way of dealing with negation “\+”. We show how badly Progol performs when dealing

with negation in chapter 9.

In order to overcome this problem, we altered the translation suite to remove the negation sign generated in SICStus and add “**not_**” to the predicate name along with the other properties. For instance, if **pred3** is used in both forms (positive and negative), GH will have both `:- modeb(*, not_pred3(-suspect))?` and `:- modeb(*, pred3(-suspect))?` in mode declarations. The same rule applies for the examples of negated predicates. Similarly, Weka can benefit from the addition of **not_** in the predicate name along with all the possible combinations that can happen in the predicate as mentioned in §4.5.3. For instance: `@attribute not_pred6 {1, 2, 12}.`

However, the presence of negated predicates in a DIP will create some problems while solving DIPs which we will discuss in detail in the chapter 7. As mentioned earlier, GH represents the problems in FOL and in particular SICStus Prolog syntax. Negation or “**not**” may cause problems in Prolog and should be used with care. “**not**” in Prolog is not exactly the same as negation in mathematics. Negation in prolog is described as:

```
not P:- P, !, fail.
```

```
not P.
```

While processing a **not** goal, Prolog tries to prove the opposite and the goal succeeds in case the opposite fails [17]. This is due to Prolog reasoning with a *closed-world-assumption*, which means that if some information is not in the background knowledge or cannot be derived from the background knowledge, Prolog assumes that it is not true and hence, its negation is true. In short, Prolog assumes negation as failure.

Some of the problems will be sorted with the use of “**\+**” instead of “**not**” in the newer versions of Prolog such as 3.12. However, the problem remains the same with uninstantiated goals. There have been some methods to overcome this problem to some extent such as *Constructive Negation* explained in [13, 14]. However, we need to adjust this problem in order to fit properly into our DIPs.

Imagine a DIP containing this piece of information: `hates(suspect1, suspect2).` A detective

may want to know which suspects do not hate each other or at least there is no evidence of them hating each other, hence `not(hates(X,Y))`. In this case, negation in SICStus does not work due to uninstantiated query. We designed GH in such a way to capture all the suspect domains during the generation process and to keep the answer to such queries. During the generation, GH automatically extracts the suspects that appear in the predicates and stores them respectively. Therefore, in the case of a negated query, GH outputs all the possible combinations where the truthfulness of the query itself (non-negated) cannot be inferred based on the background knowledge. Part of the SICStus Prolog code for implementing negated queries is shown in figure 5.9.

```
negation(Pred):-
    find_domain(Pred,Domain),
    find_domain_combinations(Domain,Combs),
    functor(Pred,P,_),
    check_not_provable_combs(P,Combs,Results),
    show(Results).
```

Figure 5.9: Partial Prolog code – negated queries

In figure 5.9, `find_domain` will capture the domain of variables appearing in the predicate “Pred”. For instance, in the case of `hate(X,Y)`, variables matching `X` and `Y` in the background knowledge will be stored. `find_domain_combinations` will output all the possible combinations of the domain variables, say, `[scarlett, green]`, `[plum, mustard]`, etc. Having all the possible combinations, GH only needs to check which of combinations can be proved by the background knowledge and subtract them from the rest of the combination. Therefore, the remaining list of combinations is the answer to the negated query. `check_not_provable_combs` performs the task of subtracting the provable combination from the rest. Part of this function, written in SICStus is as follows:

```
check_not_provable_combs(P,Combs,[H|Rest]):-
    select_element(H,Combs,RestDomain),
    Pred=..[P|H], \+ Pred,
    check_not_provable_combs(P,RestDomain,Rest).
```

5.4 Conclusions

We mentioned that GH aims to design DIPs that represent real life investigation scenarios. In order to do so, it was needed to generate past cases with less similarity to the current case. In this chapter, we discussed the main reasons that inspired us to modify the generation process. We took into account the factors such as the hierarchical importance of the data. A stochastic approach for prioritizing the information was described. We introduced formulae for ranking and weighting the predicates and suspects along with an example of a DIP generated based on those formulae. Finally, we described the shortcomings of applying negation in logic programs and the need for the addition of negation to our DIPs in order to have more realistic scenarios.

In chapter 9, we will show how the performance of GH can be deteriorated while applying the reality factors to the generation algorithm and will show how the mentioned factors help generating more realistic scenarios. The methodologies that GH harnesses in order to tackle the difficulties caused by reality factors will be discussed in chapter 7 (§7.4).

Chapter 6

Solving Preliminary Case Studies

6.1 Introduction

In chapter 3 (§3.2), we introduced a type of investigation problem akin to real life scenarios that are amenable to various reasoning techniques. We discussed the Aunt Agatha problem in §3.3 and showed that such *whodunnit* problems, where the identity of the murderer is revealed in the light of information gathered through time steps, can be tackled using a combination of machine learning, automated theorem proving and constraint solving. Cluedo-style problems, discussed in chapter 4 (§4.3), were another example to confirm that such ill-formed investigation problems are susceptible to a combination of reasoning systems. We also introduced an algorithm for generating Cluedo style problems with various levels of difficulty in §4.3.

By generalizing Aunt Agatha and Cluedo-style problems, we formally defined a type of hybrid AI problem that we named Dynamic Investigation Problems (DIP). With the use of this formal definition, we showed that we can express a variety of investigation problem where there is a set of suspects (suspected criminals or suspected illnesses) and the problem is to correctly identify the guilty party in the light of changing information. Even though we defined and introduced algorithms for generating DIPs in chapters 3 and 4, in this chapter we try to show the solution methodologies employed in tackling the preliminary case studies of Aunt Agatha and Cluedo problems. Solving these problem were among the first attempts at investigating

ill-formed problems that neither of the standalone AI techniques such as constraint solving, machine learning or automated theorem proving could solve. In this chapter, we will discuss the techniques we harnessed to tackle the Aunt Agatha and Cluedo-style problems and later in chapter 7, we will explore the DIP solving techniques in detail.

6.2 Aunt Agatha Problem

In chapters 2 and 3, we stated that one of the aims of this project is to implement a system that employs disparate reasoning techniques and to show that such a system has much potential, particularly, in scenarios where the background information is incomplete, i.e., the background information is not sufficient for solving the problem. We argued that the dynamic investigation problems that we aim to tackle, are amenable to different reasoning techniques such as constraint solving, machine learning and automated theorem proving. As mentioned in the chapter 3 (§3.3), we started building our system by looking into a whodunnit problem in the TPTP library named “*who killed Aunt Agatha*” [121]. Throughout the background and generation chapter, we showed how we started from Aunt Agatha and Cluedo to generating a generic type of DIP. Mutilating Aunt Agatha [107] was our first attempt to employ disparate AI systems to solve a problem that cannot be solved by any of those standalone systems. Recalling from chapter 3, the problem is:

Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler. Who killed Aunt Agatha?

We mutilated this problem by removing the axiom “no one hates everyone”. With this tweak, neither SICStus nor OTTER could solve the problem. With the addition of case studies, Progol was also unable to learn the profile of the guilty party. In the following sections, we will explore

how different skilled AI reasoning systems were used in concert to solve the mutilated Aunt Agatha problem.

6.2.1 Solving the Aunt Agatha Problem: HR and OTTER

By employing the HR discovery system, discussed in chapter 2 (§2.6), using *exists* as its unary production rule and *forall*, *negate* and *compose* as binary production rules, we could recover the crucial missing axiom in the mutilated Aunt Agatha. We provided HR with concepts such as “suspect” and “hates” along with the examples of those concepts. As explained in the §2.6, HR forms conjectures about these concepts by modifying and combining existing concepts. The missing axioms were included in the HR output as follows:

```
otter_format: all a b (hates(a, b) -> (-(all c (hates(a, c)))))
```

This axiom is the same as “no one hates everyone”; considering the right hand side of the implication, moving $\neg$ inwards and De Morgan’s law (inc. quantifier version): $\forall a(\exists c \neg \text{hates}(a, c))$. In figure 6.1, we present a subset of HR conjectures from which the missing axiom was extracted.

By entering the background knowledge and the missing axiom in the OTTER syntax, we finally specified six conjectures to the OTTER theorem prover. The axioms represented the information from the problem statement and the conjectures were respectively: Agatha killed/didn’t kill Agatha; Butler killed/didn’t kill Agatha; Charles killed/didn’t kill Agatha. OTTER successfully proved that Agatha killed Agatha, and that Butler and Charles didn’t kill Agatha. It failed to prove any of the other conjectures. The process is explained in detail in chapter 3 (§3.3) and [107] and the OTTER proof can be found in appendix B.1.

6.2.2 Solving the Aunt Agatha Problem: SICStus Prolog

As a high level declarative programming language, SICStus Prolog provides a powerful suite based on a subset of FOL and yet, as showed earlier in chapter 3, it could not solve the mutilated Aunt Agatha due to the following missing rule:

```

((exists b (hates(a, b))))
(-(all b ((hates(a, b)) -> (hates(b, a)))))
all b (((hates(a, b)) -> (-(all c ((hates(a, c)) -> (hates(c, a)))))))
((exists b (hates(a, b) & -(hates(b, a)))))
all b (((hates(a, b) & (all c ((hates(a, c)) -> (-(hates(c, a))))) ->
  (-(hates(b, a)))))
(-(all b ((hates(a, b) & -(hates(b, a))) -> (hates(b, a)))))
(((all b ((hates(a, b)) -> (-(hates(b, a))))) -> ((all c ((hates(a, c)
  & hates(c, a)) -> (-(hates(c, a)))))
(((all b ((hates(a, b) & hates(b, a)) -> (-(hates(b, a))))) -> ((all c
  ((hates(a, c)) -> (-(hates(c, a)))))
all b (((hates(a, b)) -> (-(all c ((hates(a, c) & -(hates(c, a)) ->
  (hates(c, a)))))))
all b (((hates(a, b) & (all c ((hates(a, c)) -> (-(hates(c, a))))) ->
  ((all d ((hates(a, d) & hates(d, a)) -> (-(hates(d, a)))))
all b (((hates(a, b) & (all c ((hates(a, c) & hates(c, a)) ->
  (-(hates(c, a))))) -> ((all d ((hates(a, d)) -> (-(hates(d, a)))))
all b (((hates(a, b) & -(hates(b, a)) & (all c((hates(a, c)&hates(c,a)
  -> (-(hates(c, a))))) -> ((all d ((hates(a, d)) -> (-(hates(d, a)))))
all b (((hates(a, b) & hates(b, a)) -> (-(all c ((hates(a, c) &
  hates(c, a)) -> (-(hates(c, a)))))

```

Figure 6.1: A subset of HR output

```
killed(X, Y) :- hates(X, Y), \+ richer(X, Y).
```

Butler and Agatha, were both answered as the killer. In order to clarify, let us look at the following axioms:

```

hates(agatha,agatha).   hates(butler,agatha).   :- hates(charles,agatha).
hates(agatha,charles).  hates(butler,charles).   :- hates(charles,charles).
:- hates(agatha,butler).
```

Figure 6.2: Partial background knowledge from Aunt Agatha problem

According to the axioms in figure 6.2, the problem does not express if Butler hates himself. However, based on Prolog closed-world assumption, Prolog deduced that Butler does not hate himself “:- hates(butler,butler).” In other words, eliminating the axiom “no one hates everyone” did not influence the solution. However, SICStus could not yet rule out Butler from the list of murderers. Since it is not mentioned that Butler is richer than Agatha in the background knowledge, Butler satisfied the second condition for being the murderer:

`\+ richer(butler,agatha).` Therefore, Butler was deduced to be the murderer along with Agatha.

The solution can be changed with the way we interpret the sentence “Butler hates everyone who is not richer than Agatha”. The Prolog translation of this sentence is:

```
hates(butler,X):- \+ richer(X,agatha).
```

In order to eliminate Butler from being the murderer, we needed to tweak the problem statement. Considering logical contraposition, we replaced the above sentence by its contrapositive equivalent:

```
richer(X,agatha):- \+ hates(butler,X).
```

From this statement, SICStus deduced: `richer(butler,agatha).` Since Butler does not hate himself, he is richer than Agatha and consequently cannot satisfy the second condition for being the murderer. Therefore, Butler was ruled out from being the murderer. With this example, we realized that by direct translation of the Aunt Agatha problem description, SICStus could not solve the problem unless a slight modification was enforced to one of the axioms in the background knowledge.

6.2.3 Solving the Aunt Agatha Problem: Progol and Prolog

As explained in chapter 2 (§2.8.4) and [110], Progol is a state of the art ILP system used to learn from examples. We employed Progol to learn a set of rules about the murderer and in order to do so, we initially needed to mutilate the problem by removing the axiom about the murderer in the current case: `“killed(X, Y) :- hates(X, Y), \+ richer(X, Y).”`. We provided Progol with two sets of previously solved cases; each of the cases were comprised of a set of positive and negative examples. The solved case studies were constructed in such a way that (a) there was a direct mapping from Agatha, Butler and Charles in the current case to one of the people in the case studies, where the Agatha character was always killed and (b) all the axioms from the problem statement about who could possibly hate who, etc., were upheld. This means that the people corresponding to Agatha, Butler and Charles in the case studies, held the same relationships that the main characters satisfy in the current case.

In the first set of past cases, the data reflected the fact that the murderer and the victim were always the same person. Part of the data is shown in figure 6.3. Progol hypothesized the rule

```
hates(bob, kate).
hates(bob, dan).
:- richer(bob, bob).
murderer(bob).
victim(bob).
```

Figure 6.3: The murderer and the victim are the same

“`murderer(X):- victim(X)`” or depending on how we defined the mode declarations and the murderer/victim predicate in the case studies, Progol produced: “`murderer(X,X)`”.

In the second set of past cases, the examples were designed to still satisfy the problem axioms but we changed the murderer/victim combinations. Part of the data is shown in figure 6.4.

```
:- richer(b1, c1).
hates(b1, c1).
murderer(b1,c1).
:- richer(a2, c2).
hates(a2, c2).
hates(a2, b2).
murderer(a2,c2).
```

Figure 6.4: The murderer and the victim are *not* the same

Progol hypothesized “`murderer(X,Y):- hates(X,Y), \+ richer(X,Y)`”. Considering that our examples were a collection of previously solved cases similar to Aunt Agatha, we subsequently fed the hypothesized rules along with the problem description to SICStus. With either of the above hypotheses, SICStus was able to rule out Butler and Charles and highlighted Agatha as the only murderer.

6.2.4 Solving the Aunt Agatha Problem: HR and CLPFD

We argued in chapters 2 and 3, that converting our problems into the syntax of various AI problem solving systems, will at least enable us to compare different AI techniques. It can also enable us to use combination of techniques for solving complicated problems. Even though the DIPs are represented in SICStus Prolog syntax, the embedded constraints are not expressed in

CLPFD format. Constraint logic programming over finite domains was explained in chapter 2 (§2.5.1). In the system introduced in [23] and explained in §2.5.2, Charnley and Colton managed to express general problems in the syntax of CLPFD. With the use of that system, we were able to convert the Aunt Agatha problem into the CLPFD syntax and in the next chapter, we will explore how DIPs are represented in that format. In the mutilated Aunt Agatha problem, we used HR to recover the missing information and used the CLPFD translation suite to convert the recovered axiom along with the background knowledge to CLPFD format. For instance, let us consider the axiom “*Charles hates no one Aunt Agatha hates*”. In OTTER syntax, this axiom is: `all x (hates(agatha,x) → (¬ hates(charles,x)))`. Considering the tautology $(A \rightarrow B) \longleftrightarrow (\neg A \vee B)$, we can replace the implication and rewrite the axiom as: `all x ((¬(hates(agatha,x))) | (¬(hates(charles,x))))`. The automatic translation of this axiom is shown below: (the full translation of the Aunt Agatha problem can be found in appendix B.1.3)

```
% all x ((¬(hates(agatha,x))) | (¬(hates(charles,x))))
p_4_1(MT,MT1):-
  sub_sets_with_types([1],S),
  p_4_2(S,MT,MT1).

p_4_2([],MT,MT).
p_4_2([[V1]|Ss],MT,MT2):-
  p_4_3(V1,MT,MT1), % (¬(hates(agatha,x))) | (¬(hates(charles,x)))
  p_4_2(Ss,MT1,MT2).

p_4_3(V1,MT,MT2):-
  p_4_4(V1,MT,MT1,RV1), % ¬(hates(agatha,x))
  p_4_5(V1,MT1,MT2,RV2), % ¬(hates(charles,x))
  RV1 #\ / RV2.

p_4_4(V1,MT,MT,RV0):-
  pred_var_at_typed(hates,[0,V1],MT,PV1),
  PV1 #= 0 #<=> RV0.

p_4_5(V1,MT,MT,RV0):-
  pred_var_at_typed(hates,[2,V1],MT,PV1),
  PV1 #= 0 #<=> RV0.
```

Figure 6.5: Part of the automatic translation of the Aunt Agatha problem to CLPFD

The details of the translation process are mentioned in chapter 2 (§2.5.2). The CLPFD trans-

lation suite [23, 25], initially, takes each axiom as a string and tokenizes and, subsequently, parses it using the Definite Clause Grammar. In the Aunt Agatha problem, the system identifies four predicates, *killed*/2, *hates*/2, *lives*/1 and *richer*/2 and three constants as *agatha*, *charles* and *butler*. It then tries to find all the possible combinations of constants for each predicate and assigns boolean variables to them. For instance, there would be 9 boolean variables representing all the possible combinations of *hates* and another 9 for *richer*. In the above translation, shown in figure 6.5, `p_4_1`, takes the list of possible value combinations and `p_4_2`, recursively goes through the list and calls the *sub-expression* predicate “`p_4_3`”. Since the axiom “`all x (hates(agatha,x) → (¬ hates(charles,x)))`” is a combination of two sub-expressions, predicate `pred_4_3`, itself, calls two more predicates representing the sub-expressions: `¬hates(agatha,x)` and `¬hates(charles,x)`. Consequently, the reification variables, say `RV1` or `RV2`, check whether the constraints hold or not. With the help of the translation suite, we managed to represent the Aunt Agatha in CLPFD format and the solution is as follows:

```

hates(agatha,agatha):1
hates(agatha,charles):1
hates(butler,agatha):1
hates(butler,charles):1
killed(agatha,agatha):1
lives(agatha):1
lives(butler):1
lives(charles):1
richer(butler,agatha):1

```

Figure 6.6: Aunt Agatha CLPFD solution

The constraints that were satisfied are shown in figure 6.6. It is clear that only the combination “`killed(agatha,agatha)`” was satisfied and none of the other combinations that were tested in the predicate *kill* could fulfill the constraints.

We mentioned earlier in this section that by translating a problem into the syntaxes of various AI systems, we can compare the performance of the respective systems or even use a combination of systems in order to tackle more sophisticated problems. The constraint translation suite is a powerful technique particularly in the algebraic domain as discussed in [23]. However, in the

next chapter we will discuss why using the translation suite is not practical for DIPs. In the following sections, we examine a combined reasoning technique for a board game known as Cluedo [2].

6.3 Cluedo-style Problems

Recalling from chapter 4 (§4.3), in Cluedo [2] board game, the player moves around the mansion with nine rooms where the murder can take place. The player collects clues to infer about the suspect that has committed the murder and of course, the murder weapon. As showed in §4.3, the information gathered by the player in each room can be formulated as suspects, grounded facts (predicates) and constraints, while each room can be interpreted as a time step; the same constituents of a DIP. We chose Cluedo due to its straightforward formulation to DIP scenarios. We discussed in §4.3 that GH's generation module yields more complicated Cluedo-style scenarios due to additional constraint, facts and suspects. The number of suspects, relations between suspects (predicates) and the rules that are defined in a GH generated scenario, exceeded the number of their counterparts in the original Cluedo game in order to present a more complicated problem. GH employs HR, SICStus Prolog and Progol to solve Cluedo-style problems.

6.3.1 Solving Cluedo-style problems: HR

We mentioned earlier that if the background knowledge of a Cluedo scenario is not sufficient to solve the mystery, previously solved cases can hopefully help us to learn a set of rules about the mystery. For instance, *the murderer should be tall and left-handed*. These rules can ultimately replace the missing information in the current case or at least make the current case easier to be solved. We discussed the Cluedo generation algorithm in chapter 4 (§4.1) and we showed an example in §4.3.1. Using the DIP formal definitions, a Cluedo problem is comprised of `:= < suspects, constants, facts, constraints, case_studies >` in which all the elements, except the case studies, are constituents of the *current case*.

We employed the HR automated theory formation system, in a similar way to how we did with the Aunt Agatha problem, in order to solve Cluedo mysteries. However, instead of recovering the missing axiom in Aunt Agatha, we needed HR to form a theory about the previous case studies. HR discovering axioms about the case studies enabled us to pick one suspect as the answer or at least reduce the number of suspects. An example of a Cluedo scenario is shown in figure 6.7 on page 131.

The HR running configurations will be discussed in chapter 8. We also omit the details of HR production rules as they were mentioned in §6.2. We used HR's `forall`, `exists`, `negate` and `compose` production rules in 1000 steps. HR produced 433 conjectures. We were interested in the ones that were explaining the *murderer*, i.e., conjectures that contained the word murderer, either in the body or the head of the clause. Consequently, the conjectures were used along with the current case of the Cluedo scenario in the SICStus Prolog solver. HR conjectures are shown in figure 6.8. In the next chapter, we will explain how the conjectures are automatically chosen to be used in the solution process. Before that, we need to recall the translation suite (TS) from chapter 4 (§4.5).

A note on the Translation Suite (TS):

As discussed in chapter 4 (§4.5), the translation suite (TS) used in the generation module, has capabilities that can also be used during the solution. In the generation module, TS translated the problem into the syntax of various programming languages and AI tools. In this part of the project, GH uses TS to (a) perform slight modifications to HR conjectures and (b) extract the rules, negations and facts in the current case.

Even though HR has the functionality of outputting the conjectures in Prolog, TPTP or some other formats, slight modifications are required to automatically feed those outputs to different AI solvers. These modifications are made with the help of TS. Let us consider one of the HR conjectures:

```
410: is_short(A,B) :- case_study(A), suspect(B), \+ is_tall(A,B).
```

It is clear that the id_number of conjectures, (410) in this example, should be eliminated. TS also considers the predicates that contain more than one arity where the first arity indicates the

```

% at Time = 1

fact(1):- angry(mustard).
fact(2):- was_found(candlestick,diningroom).
fact(3):- potential_murder_weapon(spanner).
fact(4):- is_tall(peacock).
fact(5):- was_found(revolver,patio).

% at Time = 2

fact(6):- murder_scene(livingroom).
fact(7):- angry(peacock).
fact(8):- was_found(ropes,kitchen).
fact(9):- is_short(mustard).
fact(10):- is_tallest(green).

% at Time = 3

fact(11):- loves(mustard,scarlett).
fact(12):- suspect_room(livingroom).
fact(13):- is_tall(green).
fact(14):- angry(white).

% at Time = 4

fact(15):- is_short(scarlett).
fact(16):- angry(green).
fact(17):- is_tall(plum).
fact(18):- murder_weapon(spanner).
fact(19):- is_short(white).

% at Time = 5

fact(20):- missing_weapon(dagger).
fact(21):- loves(white,scarlett).
fact(22):- is_shortest(scarlett).

% at Time = 6

fact(23):- was_found(leadpipe,spa).
fact(24):- loves(peacock,scarlett).
fact(25):- missing_weapon(spanner).
fact(26):- suspect_room(theater).
the_murder_weapon(A) :-
    \+was_found(A,B),
    missing_weapon(A),
    potential_murder_weapon(A).

```

Figure 6.7: A Cluedo scenario

```

murderer(B):-    angry(B), is_tallest(B).
murderer(B):-    is_tall(B),  loves(B,C).
murderer(B):-    is_tall(B),  loves(B,C), angry(C).
murderer(B):-    is_tallest(B), loves(B,C).
murderer(B):-    is_tall(B),  loves(B,C), loves(D,C).
murderer(B):-    is_tall(B),  loves(B,C), loves(B,D).
murderer(B):-    loves(B,C),  \+ (is_short(B)).

```

Figure 6.8: A part of HR conjectures

case number. For instance, in the predicate `is_short(A,B)`, the arity `A` is removed by TS as it trivially indicates the case number. The addition of case numbers was due to the initial thought that they are important to be specified in Progol and HR. Case numbers were later removed during the generation of past cases as shown in appendix A.2. However, TS still benefits from a function that checks the predicates in the conjectures generated by HR, Progol, Weka, etc., and compares them with their corresponding predicates in the current case scenarios. Overall, TS has the ability to modify the syntax of information to be suitable for the systems involved in the solution.

Recalling from §5.3 in chapter 5, GH adds negated facts to the problem description in order to make the problem more realistic. We also discussed why it is important for GH to be able to deal with the “negation” sign in the correct way. The drawbacks of negated queries in logic based solvers were also discussed. In the solution process, TS is needed to deal with negation. TS’s task is to extract predicates with `\+` and to replace them with the keyword “negation”. As discussed, the replacement is essential while defining the Progol mode declarations. TS executes the same negation functions showed in figure 5.9, only this time, on the conjectures generated by the various machine learning systems such as HR, Progol and Weka. Figure 6.9, depicts the process of replacing negation. Upon identifying the rules, TS, finds the rules that contain the negation sign and replaces it with the word “negation”.

```

check_for_negation(H,Head):-
    H=..[:-,Head,Body],
    Head=..[_|HeadArgs],
    include_variable(HeadArgs),
    Body=..['',BodyElem1,BodyElem2], BodyElem1=..[Sign,BodyElemBody],
    Sign==(\\+)->
    Body2Elem1=..[negation,BodyElemBody], Body2=..['',Body2Elem1,BodyElem2],
    H2=..[:-, Head,Body2],numbervars(H2,23,_),
    portray_clause(H2);
H=..[:-,Head,Body],
Head=..[_|HeadArgs],
include_variable(HeadArgs),
numbervars(H,23,_),portray_clause(H).

```

Figure 6.9: Partial Prolog code for TS – negated queries

6.3.2 Solving Cluedo: HR and Partial Solver

We showed a segment of HR produced conjectures in figure 6.8 and explained how TS trims the HR conjectures to be ready for the solver. Recalling from §6.2.3 and §6.2.4, the background knowledge along with the conjectures could be either fed to the Prolog solver or can be translated to the syntax of constraint logic programming, as showed in figure 6.5, to be consequently tackled by the CLPFD solver. However, the mentioned methodologies have shortcomings that we intend to explore in this section.

Let us consider the example showed in figure 6.7. The Cluedo scenario along with HR produced conjectures, shown in figure 6.8, were used to yield an answer. For instance, by feeding the current case into the SICStus Prolog solver, along with the conjecture:

```
murderer(B):- is_tall(B), loves(B,C).
```

the following result was obtained:

```
'White' has killed the victim by 'CandleStick' at 'LivingRoom'.
```

The same result is produced by the CLPFD solver by translating the background knowledge and the conjecture into the CLPFD format. The translation yielded by the translation suite and solution produced by SICStus CLPFD module can be found in appendix B.2. We will explain why in the following scenarios, the CLPFD solver cannot be useful:

- When none of the suspects satisfy the conditions of being the answer to the problem
- When the ill-formed background knowledge is not sufficient to infer one or few of the suspects as the ultimate answer

Recalling from chapters 4 and 5, it may happen that neither HR (nor any of the standalone AI systems that may be used), can form a theory about the previous case studies that fully explains the current case. In addition, the information in the case studies may not exhibit the same relations between the suspects, as explained in the DIP specifications (§3.5). In such scenarios, none of the suspects in our current case can satisfy the conditions essential to solve the problem. On the other hand, we already discussed that DIPs are hybrid dynamic AI problems where the background information keeps changing over time. GH's task, encountering the mentioned scenarios, is to solve a DIP (a Cluedo problem in this case) when knowledge about the problem is not sufficient to come up with a single answer. The CLPFD translation suite and solver, as explained in §2.5.2, cannot be employed to tackle such scenarios as they are only classic constraint solvers; by classic constraint solver we mean the solver that only considers a constraint being satisfied or refuted. Therefore, we enforced a methodology that can prioritize the suspects according to their likelihood degrees.

Recalling from chapter 3, the likelihood degree of a suspect, according to formula 3.6, is defined as:

$$x' = \frac{\text{Number of constraints satisfied by } x}{\text{Total number of constraints}}$$

The constraints can be broken into their constituent predicates. The solution obtained from this method is named a “*Partial Solution*”. The GH solution module initially segments each constraint into its head and body predicates. In case of a suspect not satisfying a constraint, the number of predicates the suspect can satisfy in the body of a constraint will be counted as a weight. This weight indicates the percentage of a rule satisfied by a suspect. For instance, in the axiom: `murderer(X):- is_tall(X), loves(X,Y)`, if a suspect satisfies only one of the predicates in the body, say, `is_tall(X)`, then the likelihood of that suspect being guilty is 50%.

Let us consider the example showed in figure 6.7 on page 131, where the information about the Cluedo mystery was obtained during the 6 time steps. A subset of solutions at different time steps is shown in Table 6.1, page 136. Given the time steps 1, 2 and 3, the table depicts that Mustard and Green along with Peacock have higher likelihood degrees of being the murderer. However, these two suspects are replaced by Plum and White, considering the information at time steps 4,5 and 6. It is also clear that taking all the information into account, Peacock is chosen as the only solution. Therefore, GH employs a dynamic methodology which tackles the problem based on the information gathered at each time step. This methodology is well suited for the DIPs; the solution gets updated as the information changes at different times. This method can provide us with a partial solution where there is no sufficient information to find a complete answer. We should also bear in mind when there is no adequate evidence for a suspect as a guilty party, the solution module assigns a likelihood of zero to that suspect; and there will be no answer if all suspects have zero likelihood.

6.3.3 Dealing with different answers

Even though the partial solution methodology, guarantees a solution to the problem when there is sufficient information, there could be other solutions to the problem as well. This is due to various conjectures produced by HR or any other involved AI systems. In the next chapter, we will explain the details of all the AI systems encompassed by GH. However, suffice it to say that each of these systems may produce different hypothesis about the case studies and hence different answers can be derived. Therefore, a difficulty for GH is to choose which one of the rules will result in better solutions and what we mean by better solution. Solution “A” is better than solution “B” if it has lower *error rate*; we will discuss the methodology for calculating the error rate in chapter 9. However, for the time being, let us consider the same example mentioned in figure 6.7 and with HR’s conjectures in figure 6.8. Various solutions to the problem, given different conjectures, are shown in table 6.2 on page 137. It is clear that the same information extracted from a set of time steps can yield different answers in case of a different conjecture being chosen. For instance, Mustard has a higher likelihood degree of being the murderer (66

Current Case	Rules and Answers
T= 1,2,3	Rule: murderer(B):- is_tall(B), loves(B,C).
	No Complete Solution Found; Partial Solution as below: 50 % Mustard is murderer 50 % Green is murderer 50 % Peacock is murderer 0 % Plum is murderer 0 % Scarlett is murderer 0 % White is murderer
T= 4,5,6	Rule: murderer(B):- is_tall(B), loves(B,C).
	No Complete Solution Found; Partial Solution as below: 50 % Plum is murderer 50 % White is murderer 50 % Peacock is murderer 0 % Mustard is murderer 0 % Scarlett is murderer 0 % Green is murderer
T= 1,2,3,4,5,6	Rule: murderer(B):- is_tall(B), loves(B,C).
	Peacock has killed the victim by Spanner at LivingRoom

Table 6.1: A subset of solutions

%), when T=1,2,3, at the bottom of the table comparing to the its previous status (33 %). On the contrary, Reverend Green's possibility of being the murderer decreases.

Overall, it is inevitable that we must deal with a large number of hypotheses about case studies while employing various AI tools such as Weka, Progol, etc. One reason is due to the nature of

Current Case	Rules and Answers
T= 1,2,3,4,5,6	Rule: murderer(B):- is_tall(B), loves(B,C).
	Peacock has killed the victim by Spanner at LivingRoom
T= 1,2,3,4,5,6	Rule: murderer(B):- angry(B), is_tallest(B).
	Green has killed the victim by Spanner at LivingRoom
T= 1,2,3	Rule: murderer(B):- is_tall(B), loves(B,C), angry(C).
	No Complete Solution Found; Partial Solution as below: 66 % Green is murderer 66 % Peacock is murderer 33 % Mustard is murderer 33 % Plum is murderer 33 % Scarlett is murderer 33 % White is murderer
T= 1,2,3	Rule: murderer(B):- is_tall(B), loves(B,C), loves(B,D).
	No Complete Solution Found; Partial Solution as below: 66 % Mustard is murderer 33 % Green is murderer 33 % Peacock is murderer 0 % Plum is murderer 0 % Scarlett is murderer 0 % White is murderer

Table 6.2: Cluedo solutions – different rules & times

AI systems that GH harnesses to learn from past cases. For instance, in Progol, the simplest answers are chosen according to the Occam's Razor principle; often the hypotheses are more specific, say in Weka, due to the search algorithms it uses. Each AI system chooses a set of hypotheses based on the criteria, such as predictive accuracy that they exhibit on the past cases. However, we should bear in mind that DIP past cases are contrived with embedded information that is irrelevant to the current case scenarios. Recalling from chapter 4, page 86, the combination of randomness and the mapping function in the generation algorithm can produce scenarios that are not the same as the current case. Sometimes a more specific conjecture that does not cover all the examples in the past cases is better and sometimes vice versa. We will compare and explore the results of different conjectures and answers in chapter 9.

6.4 Conclusions

In this chapter, we focused on our preliminary analysis of the two case studies: Aunt Agatha and Cluedo scenarios. We employed a combination of reasoning techniques to tackle the ill-formed scenarios and demonstrated how a combined reasoning approach can improve the effectiveness of the solution module. Both case studies were tackled by combining two or more of the various AI systems such as HR, Progol, Prolog and the CLPFD solver. While none of the mentioned systems could solve the ill-formed problems, a combination of them revealed the effectiveness of such an approach.

As discussed earlier, in real-life scenarios, the previous cases studies of an investigation problem like a murder case, may not be completely analogous to the current case; the number of suspects may vary, the motives or murder weapons may be different or the alibi may indicate other narrations. We also explained the methodologies we used to enable the GH generation module to produce problems similar to real life investigation problems. Due to this, the hypothesis produced by machine learning systems such as Progol and HR, that are employed to discover useful information about the case studies, may not correctly and completely explain the current

case scenarios. On the other hand, due to the ill-formed nature of the current case and the lack of some axioms in the background knowledge, it may be that none of the suspects satisfy all the required constraints explaining the solution and hence no answer can be obtained. Due to this, we introduced a way to achieve partial answers based on the number of constraints or the number of constituent predicates of the constraints each suspect satisfies.

We also discussed the importance of conjecture selection in the solution module. Various machine learning systems, due to their embedded methodologies, can generate different hypothesis from which GH should choose the one that produces the most accurate likelihood degrees while applied to the current case in the solution process. The preliminary analysis paved our way to encompass a combined reasoning system in GH's solution module. The solution module will be explained in detail in the next chapter and we will discuss the ways to employ different techniques in concert to solve a DIP. We will also present a framework comprised of various machine learning systems such as Weka, HR declarative discovery system, etc., as part of the solution module. The architecture of the GH solution module describes a collection of functions, embodying a wide range of techniques, from learning hypotheses to conjecture selection using probabilistic inferences and partial solutions.

Chapter 7

A Combined Reasoning DIP Solver

7.1 Introduction

The GH system, named after the fictional drama television character Gregory House, was first introduced in chapter 4, where we explained how the generation module of GH works. Furthermore, in chapter 5, we explored the reality factors by which GH is equipped with a more realistic generation algorithm. In summary, GH’s task described in chapters 3 and 4 was (a) to automatically generate DIPs with or without the reality factors enabled and (b) to translate them into the syntaxes of various AI systems that are employed to find a solution. In chapter 6 (§6.2), we explained how to solve ill-formed whodunnit problems such as the mutilated Aunt Agatha problem, by combining disparate AI systems. We also showed how GH can solve Cluedo scenarios by employing HR and partial solver in §6.3. Those preliminary analyses were among the first attempts at investigating ill-formed problems that neither of the standalone AI techniques could solve.

Complementing the former chapter, we show here how GH harnesses various systems, in concert, for solving DIPs. We mentioned earlier that the background information of a DIP, similar to real-life investigation scenarios, might be partially available. Hence, for any standalone AI system, there might not be sufficient information (constraints, past cases and/or theorems) to be able to solve the problem. This highlights the need for developing a methodology that can

tackle such problems. Therefore, one of the goals of our work with GH is to apply distinct reasoning systems, in concert, in order to tackle such problems. In this chapter, we discuss how GH applies a combined reasoning methodology.

We will also formalize what we mean by finding a solution to a DIP and how GH proceeds when it cannot find a *single* or a set of *definite* answers. By definite answer, we mean finding the profile of the guilty party that satisfies the constraints when applied to the current case. We will demonstrate what we mean by having a solution coupled with likelihood degrees and how GH infers a set of partial solutions. In this chapter we will elucidate how GH utilizes a combination of AI systems such as HR [29], Weka [55], Progol [84], Progol+, CLPFD translator [25, 23], SICStus CLPFD [21] and *Pe-Pl* [40], in order to tackle DIPs.

7.2 GH – A Combined Reasoning System

A reasoning process which employs various problem solving techniques for solving a given task is called a *combined reasoning* process. The systems that engage such reasoning processes are called *combined reasoning systems*. These systems, in general, are supposed to perform better than their constituents. By performing better we mean that they are more efficient than their sub-systems (components) in tackling a problem. The components of a combined reasoning system can either execute the same or disparate reasoning techniques or even a combination of both. We should bear in mind that due to combining disparate systems, the combined system can have drawbacks such as inflexibility. Each of the components of a combined reasoning system may require a different representation of the problem they aim to tackle. This makes them rigid when acting jointly; that is one of the reasons we defined our problems in a generic way and chose a first order logic (FOL) representation as described in §4.4. We also implemented a translation suite for GH (§4.5) which has the ability to translate FOL problems to any of the syntaxes of the systems involved. We showed how easy the translation of DIPs to Progol is, and on the contrary, how difficult the automatic translation is, when applied to Weka syntax. Overall, GH is capable enough to deal with various syntaxes and flexible enough to cope with

an added module such as *Pe-Pl* with only a few lines of code. The procedure in which *Pe-Pl* is employed will be explained in section 7.4.

The GH system we introduced earlier is a combined reasoning system incorporating:

- The HR discovery system, explained in §2.6, which is a combined reasoning system itself, performing automated theory formation, sometimes encompassing external reasoning tools such as automated theorem proving.
- Progol, explained in §2.8.4, a machine learning system with an inductive reasoning approach.
- Progol+, explained in §2.8.4.
- Weka, a machine learning system with different algorithms for association rule mining, explained in §2.7.3.
- The SICStus Prolog CLPFD converter which translates and expresses general problems as CSPs, described in §2.5.2 and [23].
- SICStus Prolog and CLPFD library as a constraint based reasoning system, described in §2.5.1.
- *Pe-Pl* as a probabilistic reasoning approach, explained in §2.9.

As we explored each of the constituents of GH in chapter 2, these systems can be harnessed in different tasks. For instance, HR can be used to generate implied constraints [25, 35] or Weka can be used for data clustering and/or classification [15]. Throughout the following sections, we will try to justify how GH utilizes these systems according to its own requirements. First, let us consider the mechanics of the solution process.

The block diagram in figure 7.1, page 143, shows the process by which GH solves a DIP upon generating it. As shown in the figure, the process is *dynamic*; this is due to the dynamic nature of DIPs, explained in chapter 3, definition 3.9. At each time step, some segments of a DIP are revealed and GH attempts to solve the problem based on that particular piece of information.

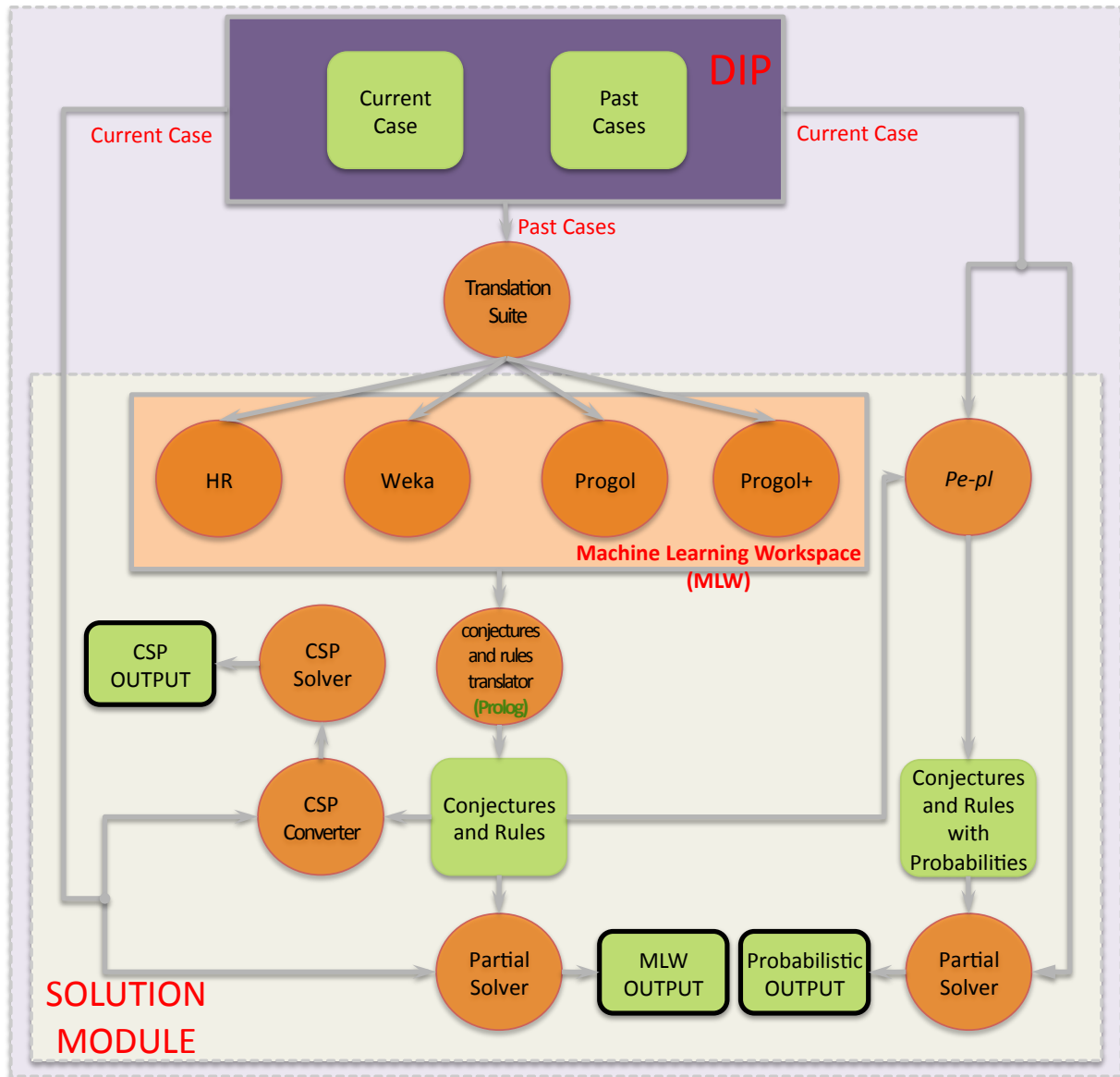


Figure 7.1: Schematic diagram of the solution process in GH

More information is obtained as time passes which may change the best solution of the problem. Hence, GH repeats the entire process until no more information is obtained.

* **Note:** Generation of the current and past cases and subsequently translating them to the syntaxes of the involved AI systems, are constituents of the GH *generation* module, as explained in chapter 4. The purpose of showing the generation module in figure 7.1 is to clarify the sequence of operations in the solution module after generating the problem.

As presented diagrammatically in figure 7.1, we have automated the following approach to

solving a DIP:

- 1) GH supplies HR, Weka, Progol, Progol+ and *Pe-Pl* with the suitable syntaxes of past cases (part of the generation module). The mentioned systems, except *Pe-Pl*, are considered as the modules of Machine Learning Workspace (MLW).
- 2) HR production rules create new concepts by modifying and combining existing concepts (predicates, suspects and constraints) presented in the past cases. HR forms conjectures about those concepts and consequently stores the conjectures.
- 3) Weka tries to find associations between predicates, suspects and constraints in the past cases through association rule mining and stores the learned rules.
- 4) Progol and Progol+ learn a set of rules about the concepts and attributes presented in the past cases through negative and positive examples.
- 5) The *Conjectures and rules translator* module shown in the schematic diagram performs extraction and translation. It extracts the rules and conjectures produced by the systems in the machine learning workspace and translates them to Prolog syntax. Ultimately, all the conjectures are input to a storage named “Conjectures and Rules”.
- 6) From this point, the solution process is split into 3 sub-processes:
 - (a) Conjectures and rules are fed to a partial solver along with the current case.
 - (b) Conjectures and rules are fed to a CSP converter along with the current case. The CSP converter translates the information into CLPFD syntax and feeds them to the CSP solver.
 - (c) *Pe-Pl* estimates the probability of the accuracy of conjectures and rules with respect to the background knowledge of the current case and stores them in the respected block: “conjectures and rules with probabilities”. These rules are fed to a partial solver along with the current case.

- 7) The output of the subsystem comprising *Pe-Pl* is “*Probabilistic Output*”. The output of the subsystem encompassing machine learning tools is called “*MLW Output*” and the output of the translated information to CLPFD is “*CSP Output*”.

In chapter 2, we explained how each of the systems, encompassed within GH, work individually. In the following sections, we will investigate how those systems are used, in concert, in order to provide solution(s) to GH.

We will show how GH extracts the conjectures with the highest predictive accuracy. Assuming that all the conjectures extracted from the MLW module have the same accuracy, GH is capable of recommending the conjectures that are more likely to be correct on the unseen example (*current case*), using the *Pe-Pl* system.

7.3 Solving DIPs

The block diagram in figure 7.1, page 143, shows the process of solving DIPs. Subsequent to the generation of the current case and the past cases, the translation suite provides the suitable syntaxes to the systems involved in the solution module. The solution module is comprised of a Machine Learning Workspace (MLW), *Pe-Pl*, translation suite and couple of different solvers. We showed the preliminary analysis of some of the MLW systems, the translation suite, the partial solver and their methodologies in chapter 6. However, we will explain them here in more detail and will also delineate the use of *Pe-Pl* in this chapter. Besides, we will elaborate on how the reality factor mentioned in chapter 5 can facilitate the solution procedure. Recalling from chapter 3, a DIP knowledge base consists of:

$$DIP := < \text{Suspects, Constants, Facts, Constraints, Case Studies, Mappings} > \quad (7.1)$$

Investigation problems are designed to model, to some extent, a generic situation which may arise in, say, medical diagnosis or solving of a crime. There are number of possible diagnosis/-suspects and the aim to use the facts and constraints in order to rank the them in terms of their

likelihood degree of being the cause of illness or guilty of the crime. We use the term *dynamic*, to describe series of such problems to be solved. At each time step t , the problem contains the information shown in 7.1 and each time step will bring to light further information that may lead to further investigation. A natural way to model DIPs is to represent them as a constraint satisfaction problems, explained in 6.2.4, where solving a DIP entails using the facts coupled with prior knowledge about the domain in the current case to narrow down the suspects to just one.

Often, however, not all the essential information is readily available and hence there will be no outright solution. As such, these problems are best modeled as partial CSPs. Also, additional relevant information can often be found in similar past cases from which the regularities can be observed and applied to the current case. The machine learning suite, shown in figure 7.1, is employed to extract the additional constraints from the case studies, i.e., when the constraints are not sufficient to narrow down the guilty suspect(s), additional information will enable us to either solve the problem or to enhance the ranking of the suspects in terms of their likelihood degree. One way to perform the ranking of the suspects, as explained in formula 3.7, is to calculate the number of constraints that are upheld by each suspect.

Recalling from the generation chapter 4, suspects, constants, facts (clauses with only one predicate in the head and empty body) and constraints that are in the form of rules (clauses with only one predicate as head and at least one predicate in the body) keep changing at different times. The case studies also change accordingly. For simplicity, in this section, we assume that DIPs are at the final stage, therefore, all the information has been gathered. GH automatically generates varied problem sets in terms of difficulty where the case studies may or may not have the information about the correct solution in the current case. By correct solution, we mean the answer that specifies the correct likelihood of guilt for all the suspects. For instance, in the former Aunt Agatha example, the correct answer was to assign 100 % likelihood of being the murderer to Agatha and 0 % to Butler and Charles. We showed in the mutilated Cluedo scenarios, that there could be a series of suspects involved in each scenario with various likelihood degrees. Hence, the task of GH is to produce a partial solution. Considering the past cases may contain information not pertinent to the current case, GH is able to at least make the problem

$A = \{a_1, \dots, a_n\}$ is a set of answers whereas:
 $a_i = \{S'_i, \dots, S'_m\}$ where m is the *sizeof*(*SuspectDomain*)
 S'_i is the likelihood degree of Suspect S_i where $S_i \in \textit{SuspectDomain}$

Figure 7.2: The set of answers: ‘A’

easier by reducing the number of suspects or by highlighting the more probable suspects, albeit not coming up with the correct answer. Considering the tables from previous chapters (6.1 and 6.2), the answer A produced by GH is shown in 7.2 (we use the terminologies introduced in chapter 3). The GH solution module shown in figure 7.1 encompasses a set of AI systems in MLW, in order to extract a set of hypotheses from the case studies so that the partial solver can produce the set of answers (A).

GH has been designed to generate DIPs in such a way that the logic program representing the background knowledge of the current case (B), along with the set of hypotheses (H) extracted from the case studies, should explain the set of answers (A), i.e., $B \wedge H \models A$. A set of constraints, C, is embedded in each case study, where each constraint is a single clause with a certain number of literals, L , conjoined in the clause and a maximum arity of A . The embedding is such that the clause is true for a set of suspects in a number of case studies. An embedded clause can be expressed, according to the terminologies explained in chapter 3, in formula 7.2. GH employs MLW to mine C_I from the case studies. Note that the embedded constraints may be other smaller conjunction of literals which can also be true for a subset of suspects. The problem is to mine the constraints which are true for a subset of suspects and contain as many literals as possible. We will explain why the lengthier constraints are more valuable in chapter 9 (§9.2.6). We will elaborate on how we evaluate the answers and calculate the error rate of GH in chapter 9.

$$C_i(X) \leftarrow P_1(-, \dots, X, \dots, -) \wedge \dots \wedge P_L(-, \dots, X, \dots, -) \quad (7.2)$$

7.3.1 GH employing HR2

Mathematics has been the major domain of investigation for the HR theory formation system. However, HR has been used to produce theories in other domains such as games, vision and bioinformatics. In general, it is possible to use HR to form theories in any domain given the background knowledge can be expressed in first order logic. Even though HR is a powerful discovery system and was efficient on the primitive instances of DIPs such as Aunt Agatha and Cluedo, it could not extract any significant information or any information at all given the case studies in generic DIPs. Recalling from chapter 2 (§2.6), HR uses a selection of pre-defined *production rules* to create new concepts by modifying or combining the existing concepts. We used *negate*, *forall*, *exists* and *compose* to motivate the GH configuration and we were hoping that HR will produce the conjectures useful for the solution of the current case. HR identifies the conjectures empirically by comparing the data-tables of different concepts. For instance, HR generates an equivalence conjecture when the data-tables of two concepts are the same or similar and generates a non-existence conjecture when there is no example in the data-table of a concept; hence, HR conjectures that the concept is inconsistent with the axioms of the domain.

We initially started the HR theory formation system on the *type-1* (simplest) forms of DIPs; various DIPs in terms of levels of difficulties will be explored in detail in chapter 9, but suffice to say, the simplest form of a DIP (type-1) contains the case studies where: (a) the predicates (facts) contain only one argument, and (b) the embedded rules are comprised of only one predicate in the body and there is only one answer to the problem. Using the definitions introduced in chapter 3 (§3.4 and §3.5), the rules and facts in each of the past cases in type-1 DIPs are shown in figure 7.3.

As explained in chapter 4 (§4.5), past cases in a DIP are translated to the format of concepts and examples understandable to HR. The domain of HR investigation in this particular task is case studies of DIP and the objects of interest of that domain are concepts: the constituents of a DIP that are specific definitions within that domain. Hence, definitions of suspects, constants and predicates each form a concept in HR. Besides, their instances are the examples of the

Facts: $P = \{P_1, \dots, P_n\}$ such that $\forall i \ 1 \leq i \leq n, \text{ Arity}(P_i) = 1$ (7.3)

Rules: $C = \{C_1, \dots, C_m\}$ such that $\forall i \ 1 \leq i \leq m$ (7.4)

The set of predicates appearing in the body of $C_i = B_{C_i}$ such that
 $B_{C_i} = \{BodyPred\}$ and $BodyPred \in P$ (7.5)

Figure 7.3: *type-1* DIP facts and rules

respective concepts. Within a concept, there are objects and/or sub-objects that satisfy the concept definition. For instance, in a DIP, “*suspects*” is a concept and the examples are, say, **s11**, **s54**, etc. We have seen the examples of HR concept definitions and their data-tables, which link the concepts to their examples, in the background chapter §2.6. We also described how TS translates DIPs to HR domain file in figures 4.7 and 4.7c. Here, we show a partial example of the HR interpretation of two of the existing concepts, translated from case studies in a *type-1* DIP to HR domain file. The concepts are **answer** and **pred7**, represented in Prolog format with “**dip2**” and “**dip9**” indicating the id of the concepts.

```
concept(dip2, [A]) :- suspect(A), answer(A).
concept(dip9, [A]) :- suspect(A), pred7(A).
```

Figure 7.4: Examples of HR concepts

The data-tables for these concepts contain the examples satisfying their definitions. HR, given the background knowledge (case studies in our project describing the concepts with their data-tables), uses a selection of pre-defined production rules iteratively to construct new concepts. These concepts are formed by modifying or combining the existing concepts. HR also uses empirical conjecture making techniques to produce conjectures by relating one or more concepts. For instance, it compares the examples of two concepts (data-tables) in order to generate equivalence conjectures. For the concepts shown in figure 7.4, the data-tables are identical. In other words, the same suspects satisfy/dissatisfy both **answer(A)** and **pred7(A)** as shown in figure 7.5. Therefore, HR makes the following equivalence conjecture between the two concepts: *forall A : suspect(A), answer(A) ↔ suspect(A), pred7(A)*. Using this equivalence conjecture, SICStus Prolog solver solved the problem.

The production of conjectures is driven by the production of concepts. Given an example of

The positives are:

s10, s15, s17, s18, s20, s21, s23, s24, s32, s34,
s38, s39, s40, s45, s47, s49, s52, s53, s54, s56

The negatives are:

s11, s12, s13, s14, s16, s19, s22, s25, s26, s27,
s28, s29, s30, s31, s33, s35, s36, s37, s41, s42,
s43, s44, s46, s48, s50, s51, s55, s57, s58, s59

Figure 7.5: Data table for concepts: dip2, dip9

a DIP, with maximum $Arity(P_i) = 3$, $SizeOf(B_{C_i}) = 3$ in 50 production rule steps, HR managed to create 26 new concepts which resulted in 21 conjectures, one of which, played a key role in solving the problem using SICStus Prolog. Table 7.1 shows the number of concepts and conjectures produced by HR in 50 production steps. As seen in figure 7.1, HR exhibited an upward trend in producing concepts and conjectures after repeatedly increasing the steps. We should note that HR constructs the simplest concepts first. By increasing the complexity of the DIPs, HR performance deteriorates in that it requires more time to come up with complicated solutions. The search complexity of HR is increased by increasing the complexity of concepts (more predicates and/or arguments in a concept).

One justification for HR's weak performance is the insufficient number of the past cases which is usually between 3 to 5. Another reason is the sporadic relations between the examples of initial concepts. For instance, recalling from the generation algorithm in §4.4.2, there might be a relation between the second argument of **pred1** and the fourth argument of **pred10**, with

	steps=50	steps=500	steps=1k
Existing Concepts	12	12	12
New Concepts	16	106	134
Total Concepts	28	118	146
Implication Conjectures	4	48	86
Non-existence Conjectures	17	140	232
Equivalence Conjectures	2	70	238
Total Conjectures	23	258	556

Table 7.1: HR Concept formation

arities 3 and 5 respectively. We also know that there could be 50 suspects in total appearing in case studies.

Figure 7.6b contains an HR produced concept with all the suspects satisfying/not satisfying the relations mentioned in the concept. It shows that only the suspects `s43`, `s47`, `s40`, `s45` satisfy the relation. Figure 7.6a contains part of a DIP scenario where the second and third rows reveal the suspects that satisfy the definition of the concept.

```
pred4(s49,s40,s41).
pred4(s47,s45,s40).
pred6(s49,s41,s43,s46).
pred6(s43,s47,s40,s45).
pred6(s43,s48,s46,s47).
```

(a) DIP Partial scenario

```
concept(s105_0,[A, B, C, D]) :- suspect(A), suspect(B),
suspect(C),suspect(D), pred6(A,B,C,D), pred4(B,D,C).
f(s10)=[] f(s11)=[] f(s12)=[] f(s13)=[] f(s14)=[] f(s15)=[]
f(s16)=[] f(s17)=[] f(s18)=[] f(s19)=[] f(s20)=[] f(s21)=[]
f(s22)=[] f(s23)=[] f(s24)=[] f(s25)=[] f(s26)=[] f(s27)=[]
f(s28)=[] f(s29)=[] f(s30)=[] f(s31)=[] f(s32)=[] f(s33)=[]
f(s34)=[] f(s35)=[] f(s36)=[] f(s37)=[] f(s38)=[] f(s39)=[]
f(s40)=[] f(s41)=[] f(s42)=[] f(s43)=[[s47, s40, s45]]
f(s44)=[] f(s45)=[] f(s46)=[] f(s47)=[] f(s48)=[] f(s49)=[]
f(s50)=[] f(s51)=[] f(s52)=[] f(s53)=[] f(s54)=[] f(s55)=[]
f(s56)=[] f(s57)=[] f(s58)=[] f(s59)=[]
```

(b) HR concept s105

Figure 7.6: DIP partial scenario and HR concept

Albeit, given our initial problem types, HR's conjecture making technique showed promising results, we will explain in chapter 9 why we mainly relied on the new version of HR in our final experiments, i.e., HR3.

7.3.2 GH employing Weka

As discussed in chapter 2 (§2.7.3), Weka is a data mining program comprised of various machine learning algorithms for classification, clustering, association rule mining, etc. GH employs

Association Rule Mining algorithms (ARM), from Weka, in order to discover information from the DIP case studies. As explained in [101], ARM enables us to learn the rules of classification and clustering of the data. On the other hand, DIPs contain a set of solved case studies, through which GH aims to learn pertinent information about the current case. Recalling from chapter 4, case studies are comprised of two sets of suspects: one forming the “*answers*” and the other indicating the “*non-answers*”. Therefore, with regard to suspects that are constituents of the answer set, we have classes of positive and negative examples. By learning the rules of this classification, GH learns the profile of the guilty party, which will be consequently used on the current case suspects, hence, the suspects most fitting the bill are the answers.

We discussed the concept of *Association Rule Mining* [101] in chapter 2 (§2.7.1) and furthermore, explored the *Apriori* algorithm [6] in §2.7.2. In chapter 4 (§4.5.3), we also introduced an algorithm for translating DIP problems to Weka syntax. In this section, we describe how GH, with the use of Apriori algorithm, learns the profile on the guilty party from the case studies.

As described earlier, the finding of association rules can be applied to a large database of transactions. For instance, in a supermarket example explained in §2.7.2, managerial decisions are based on learning the relations between a large collection of items. The manager can find which items are often purchased together and can therefore glean the relation between the supermarket’s departments. This piece of information can be useful for various purposes such as marketing which suggests the items that should be placed in close proximity.

When solving a DIP, GH tries to apply association rule mining on the past cases, hence, the items of interest are the *suspects* appearing in the case studies and the database of transactions are the *facts* presented in those cases. We should consider the major difference between a DIP and a classical association rule mining problem such as the supermarket. In the supermarket example, items of interest like *bread* or *milk*, hold binary attributes, i.e., in every transaction, they either appear or not. However, in every DIP case study, every suspect has more than 2 attribute values depending on the number of *predicates* each suspect appears in and the arity of those predicates. In other words, in the supermarket example, the occurrence of bread in a transaction can be implied by either 0 or 1, whereas, in a DIP, a suspect may appear in different

arguments of a predicate. Therefore, the attribute values associated with each suspect are the argument(s) locations in which they appear or ‘0’. For instance, if a suspect appears in the 1st and 2nd arguments of **pred2**, then its attribute values are $\{1, 2\}$. Bear in mind that the number of attribute values each suspect can take, depends on the arity of that predicate. For instance, suppose that **pred5**, with arity 3, has 2³ attribute values: $\{0, 1, 2, 3, 12, 13, 23, 123\}$; each digit representing a location and two or three digits representing a simultaneous combination of locations. Similarly, arity n yields 2 ^{n} attribute values. Therefore, each item of interest (suspect) can hold one of the attribute values.

Another important note is that in DIPs, unlike the supermarket example, GH is only interested in the attribute values rather than the name of the items itself. Considering that attribute values depict the locations in which the items appear, in every past case, similar to real-life scenarios, the name of the items of interest (suspects) change and they may be different to the current case suspects. The crucial task is to learn the profile of the guilty party where in each past case may have a different name but hopefully has the same relationship with other suspects and holds the same attribute values. This is why the name of the suspects are eliminated through the translation suite of GH. To be more clear, let us use the DIP definitions introduced in chapter 3 along with the Apriori algorithm notations in [101].

Let P be a set of predicates in a DIP case study and a set of items in the association rule mining (ARM) problem:

$$P = \{p_1, p_2, \dots, p_n\} \quad (7.6)$$

Let S be a set of suspects appearing in predicates:

$$S = \{s_1, s_2, \dots, s_k\} \quad (7.7)$$

Let F be a set of facts in a DIP case study and a set of transactions in the association rule

mining problem:

$$F = \{f_1, f_2, \dots, f_m\} \quad \text{where each fact or transaction is as follows} \quad (7.8)$$

$$f_p := \langle p, s \rangle \quad \text{where } p \in P \text{ and } s \subseteq S \quad (7.9)$$

The attribute values for each transaction f , depends on the predicate arity appearing in that particular transaction.

$$f\{p\} = \text{set of attribute values} = \{\text{all the combinations of } \text{arity}(p)\} \quad (7.10)$$

$$\text{size}(f\{p\}) = \sum_{k=0}^{\text{arity}(p)} \binom{\text{arity}(p)}{k} \quad (7.11)$$

In case a suspect appears in a transaction, f_p contains all the argument location(s) where the suspects can appear in the predicate p .

Let p' be a subset of all items (predicates) in our problem. By association rule mining, we are looking for the implications where $p' \implies p_k$ where p_k is a single item in P and p_k does not appear in p' . The algorithm in [101] shows how to extract all the rules that highlight any association between sets of items. GH employs that algorithm through Weka to run through the database and find the candidate item-sets (predicates) that their support at each tuple is above the specified threshold (*minsupport*). These candid item-sets, which form the frontier set will then be combined to possibly form the new frontier sets. The algorithm terminates when the new frontier set becomes empty.

GH can impose certain restrictions on the rule mining process. For instance, we might be only interested in rules that contain a certain item in the consequent of the rules or similarly, certain items in the antecedent. As mentioned earlier, each scenario of a case study contains the predicate ‘**answer**’ which can be specified in the imposed limitations for the consequent of the rules. Such limitations are called ‘*syntactic constraints*’ and even though GH is more interested in the rules that have ‘**answer**’ in the consequent, it extracts and outputs all the generated rules by Weka. Indeed, the main purpose is to learn the profile of the guilty party, but, as mentioned earlier in the HR solution of Aunt Agatha problem, there might be rules

that can be extracted from the case studies that do not contain the ‘**answer**’, yet can solve the problem or at least make the problem easier to be solved.

The rule mining algorithm, as explained in chapter 2, initially calculates the statistical transaction support factor, discussed in §2.7.2, for every rule. This is due to the fact that the user is only interested in the rules with a higher support than the predefined threshold. We described the support and confidence factors in §2.7.1 and we will elaborate on them further in chapter 8. There are other measures of interestingness in association rule mining apart from *confidence*, such as *leverage*, *conviction* and *lift*. GH has used all these factors while encompassing Weka rule mining. The results of using these factors will be discussed and compared in detail in chapter 9. In order to show an example of association rule mining, we set GH to generate a DIP with the following specifications: (specifications terminology as discussed in the algorithm mentioned in chapter 4 (§4.4))

$Size(ConstraintDomain) = 6, \quad Size(SuspectDomain) = 10$

$Size(PredDomain) = 10, \quad C_{Locations}$

Minimum support: 0.1 (5 instances)

Minimum metric <confidence>: 0.99

Number of cycles performed: 18

Generate rules: 1452

Elapsed time: 0.065s

answer=yes 5 ==> pred1=3 pred2=3 5	conf:(1)
pred1=3 pred6=1 5 ==> pred3=2 5	conf:(1)
pred1=3 pred2=3 pred3=2 5 ==> answer=yes 5	conf:(1)
pred2=3 pred3=2 pred4=1 pred9=4 5 ==> pred1=3 5	conf:(1)
pred1=3 pred3=2 pred4=1 pred9=4 5 ==> pred2=3 5	conf:(1)
pred2=3 pred3=2 pred4=1 5 ==> pred1=3 answer=yes 5	conf:(1)
pred1=3 pred4=1 answer=yes 5 ==> pred2=3 pred3=2 5	conf:(1)
pred2=3 pred4=1 pred6=1 5 ==> pred1=3 pred9=4 answer=yes 5	conf:(1)
pred1=3 pred9=4 answer=yes 5 ==> pred2=3 pred4=1 pred6=1 5	conf:(1)
pred1=3 pred3=2 pred4=1 pred6=1 pred9=4 5 ==> answer=yes 5	conf:(1)
pred1=3 pred2=3 pred3=2 pred4=1 pred6=1 answer=yes 5 ==> pred9=4 5	conf:(1)
pred1=3 pred2=3 pred3=2 pred4=1 pred6=1 pred9=4 5 ==> answer=yes 5	conf:(1)
pred2=3 pred3=2 pred4=1 pred6=1 pred9=4 5 ==> pred1=3 answer=yes 5	conf:(1)
pred1=3 pred2=3 pred4=1 pred6=1 pred9=4 5 ==> pred3=2 answer=yes 5	conf:(1)
pred4=1 pred6=1 pred9=4 5 ==> pred1=3 pred2=3 pred3=2 answer=yes 5	conf:(1)

Figure 7.7: A subset of Weka extracted rules

As shown in figure 7.7, Weka generates a set of rules based on the minimum support threshold

which is set to 0.1 and the confidence that indicates the strength of each rule and is set to 0.99. These values are pre-assigned and the rules that satisfy these values are only extracted. A subset of extracted rules is shown in the figure 7.7. It is clear that the size of the antecedent and consequent varies and can each contain the predicate ‘**answer**’. As explained earlier, GH does not impose the syntactic constraints which forces Weka to only generate rules containing a particular consequent such as the ‘**answer**’; rule number 482 in figure 7.7 reveals this fact. The **conf** term at the end of the line indicates the confidence of each rule. As explained in the DIP translation to Weka syntax and also earlier in this section, the importance of each suspect is in the argument location(s) they appear in each predicate. The “=” sign after each predicate name, shown in figure 7.7, depicts the argument location in which that particular suspect appears. In other words, it reveals the attribute value of each item presented in a rule. For instance, in rule number 1403, the attribute value of **pred9** is 4; therefore, only the 4th argument of **pred9** plays an important role. Extracted rules are fed to the translation suite to be slightly modified and re-written into GH syntax. A part of the translation is as follows:

```

answer(A):- pred1(_,_,A,_), pred2(_,_,A,_), pred3(_,A,_), pred4(A,_),
             pred6(A,_,_,_).
answer(A):- pred1(_,_,A,_), pred2(_,_,A,_), pred3(_,A,_), pred4(A,_),
             pred9(_,_,_,A,_).
answer(A):- pred1(_,_,A,_), pred2(_,_,A,_), pred3(_,A,_), pred6(A,_,_,_),
             pred9(_,_,_,A,_).
answer(A):- pred1(_,_,A,_), pred2(_,_,A,_), pred4(A,_), pred6(A,_,_,_),
             pred9(_,_,_,A,_).
answer(A):- pred2(_,_,A,_), pred3(_,A,_), pred4(A,_), pred6(A,_,_,_),
             pred9(_,_,_,A,_).
answer(A):- pred1(_,_,A,_), pred2(_,_,A,_), pred3(_,A,_), pred4(A,_),
             pred6(A,_,_,_), pred9(_,_,_,A,_).

```

Figure 7.8: A subset of the TS translation of Weka generated rules

7.3.3 GH employing Progol

As we explained in chapter 2 (§2.8.1), Inductive Logic Programming (ILP,) is a logic-based approach to machine learning; in other words, it is one point where logic programming and

machine learning meet [91]. ILP learns the target concepts, for instance the ‘**answer**’, by going through the examples expressed in horn clause logic. The knowledge gained by ILP systems, unlike procedural knowledge gleaned from systems such as Neural Networks [83], is declarative. Examples are expressed in the simple and explicit way of logic language in ILPs, so that the generated rules follow the same characteristics. ILP systems such as Progol [84] are suitable to be employed by GH in order to form a hypothesis about the past cases.

Machine learning approaches are useful when the purpose is to induce a correct theory from a set of examples. A correct theory, covers all the positive examples and none of the negative examples. Sometimes it is practical to have a partial theory stating some of the facts and constraints about the problem along with the examples. The partial theory given as background knowledge, can simplify the learning process. ILPs construct the general rule about the problem by making use of background knowledge. As explained earlier in §3.4, every past case of a DIP is comprised of the answers about the problem along with the a set of facts, constraints, suspects and constants. The answers form the positive examples and the rest are constituents of the background knowledge about the case studies. GH uses Progol in order to learn the profile of the guilty party in a DIP. That is, solving DIPs for Progol is a straight-forward predictive learning task where a logic program implying the target literal is to be learned.

Recalling from §2.8.4, mode declarations are needed to guide the learning process in Progol. Therefore, it is essential to specify which predicates can be in the head or in the body of the rules that are extracted from the problem. Bear in mind that in §4.5 and also in the previous chapter, page 130, we showed how the translation suite provides each module with essential information. Mode declarations are at the heart of Progol, and GH cannot expect Progol to come up with an axiom such as “no one hates everyone” as per the Aunt Agatha problem (§6.2) without fine tuning the mode declarations. This is because Progol and ILP systems in general, aim to induce hypotheses based on the positive and negative examples and the background knowledge rather than spawning concepts. The Aunt Agatha problem was reformulated and tweaked in order to be amenable to Progol, while HR came up with the key axiom without any modifications made to the problem description. Overall, HR’s power is to develop concepts with the use of production rules. This highlights one of the drawbacks of Progol where we need

to exactly specify the head and body predicates in order to bias the hypothesis' search space. As discussed earlier, mode and body declarations refer to the predicates that are presented in the background knowledge. Even though this is considered as supervision on the learning process, it could also be a restriction on the hypothesis that is to be induced from the system. As DIPs are similar to real-life scenarios, GH assumes that all the background knowledge may be relevant and hence they may appear in the final hypothesis. GH translation suite, inputs all the background knowledge predicates to the '*modeb*' declarations.

In the Prolog programming paradigm, it is normally assumed that a predicate can be called with any possible combination of instantiated arguments. However, this assumption cannot be always true as sometimes a subset of arguments is needed to be grounded before a predicate can be called. Mode declarations in Progol are designed to constrain the type of arguments (input, output or constant) that appear in the hypotheses. In the background chapter (§2.8.4) and generation chapter (§4.5), we briefly explained how the type of arguments in each predicate should be assigned in Progol with the help of the '+', '-' and '#' prefixes. Considering that DIP case studies as input to Progol, are solved past cases, the mode head declaration containing the predicate *answer* should always appear with a + prefix next to its suspect argument. The associated sign '+' indicates that the argument appearing in the head predicate is an input argument and should be instantiated before the predicate *answer* is called. Consequently, all the arguments in the body mode declarations are associated with the '-' sign. This is due to the lack of knowledge about the combination of arguments appearing in the body. In real-life scenarios, we normally have no prior knowledge about the body predicates. Hence, by assigning them to '-', we guarantee a more general approach in building a hypothesis. The process of assigning types of the arguments is a part of Translation Suite's (TS) duties.

Another informative factor in Progol is the '*recall*' of each predicate. The recall is the maximum number of solutions each predicate can hold. In other words, it is the maximum number of times each predicate succeeds given an instantiation and it can be ≥ 1 or *. This number is dependent on the predicate being determinate or non-determinate. if we know the number of solutions for any instantiation, we can assign it to this factor, otherwise, * should be allocated. In general, the recall factor is used to prevent Progol from fruitlessly searching further. In our DIPs, since we

are not sure how many answers each predicate for each instantiation may have, the Translation Suite assigns each predicate with a recall of ‘*’; the sign indicates that the recall number is unbounded, however, * is 100 by default. The ‘*’ sign guarantees the predicate can succeed between 1 to 100 number of times. In the background chapter §2.8.4, we explained the bottom clause and inverse entailment which are central concepts in an ILP system. Mode declarations for a problem in Progol are shown in figure 7.3.3.

```
:- modeh(*, answer(+suspect))?
:- modeb(*, pred1(-suspect, -suspect, -suspect))?
:- modeb(*, pred2(-suspect))?
:- modeb(*, pred3(-suspect, -suspect, -suspect, -suspect, -suspect))?
:- modeb(*, pred4(-suspect, -suspect))?
:- modeb(*, pred5(-suspect))?
:- modeb(*, pred6(-suspect, -suspect, -suspect, -suspect, -suspect))?
:- modeb(*, pred7(-suspect, -suspect, -suspect, -suspect, -suspect))?
:- modeb(*, pred8(-suspect, -suspect, -suspect, -suspect, -suspect))?
:- modeb(*, pred9(-suspect, -suspect, -suspect))?
:- modeb(*, pred10(-suspect, -suspect))?
```

Figure 7.9: An example of mode declarations in Progol

7.3.4 GH employing Progol +

After analyzing the Progol results which will be shown in chapter 9, we realized the need to use Progol in a more systematic way. By systematic, we mean, testing the performance of Progol rigorously in order to make sure the prefix used in the mode declaration produces the best possible combination in terms of results. As explained earlier in this section, we used the ‘-’ sign in order to guarantee a more general approach in building a hypothesis. However, it was essential to check the veracity of the produced results. We tuned GH to run Progol numerous times with mode declaration prefix varied between ‘-’ and ‘+’, each time on the same set of solved cases. We named this piece of wrapper code around Progol as Progol+. With use of this wrapper code, GH extracted all the hypotheses given all the possible combinations of prefix. GH also used the *leave-one-out* 1-fold cross validation and testing for measuring the predictive accuracy of the learned theory on the unseen examples. Predicate `leave/1`, which is normally used by Progol when the set of examples are small, leaves one of the examples as the test

and uses the rest of the set for training purposes. Recalling from DIP's solve cases, each case may contain one or a few of the suspects as positive examples and the rest as negative. GH then repeats the same test for the rest of the examples. With the help of Progol command '`test(File)`', GH was able to measure the predictive accuracy of all the hypotheses generated by Progol and use them along with the rest of the extracted knowledge in the machine learning framework showed in figure 7.1 on page 143. Part of the output containing the contingency table of Progol+ is shown in figure 7.3.4. In chapter 9 (§9.2.3) we will show that Progol+ revealed the best parameter setting leading to better results.

```
Contingency table=  -----A----- ~A
                   P|          5|          0|          5
                   |(    0.5)| (    4.5)|
                   ~P|          0|          45|         45
                   |(    4.5)| (   40.5)|
                   ~~~~~~
                   5          45          50
[Overall accuracy= 100.00% +/- 0.00%]
```

Figure 7.10: Part of Progol+ output

Contingency table contains the true and false positive and negative outcome labels of the classification task. Overall accuracy, precision, recall, F-measures and the hypothesis statistical testing factors can be obtained from this table [94]. In the example above, the Progol hypothesis has classified the examples with 100% predictive accuracy, i.e., the ratio of the sum of true positives and negatives to the total number of examples: $\frac{TP+TN}{P+N} = 1$.

In this chapter, so far, we explored the Machine Learning Workspace (MLW) showed in figure 7.1. We described how GH employed HR, Weka, Progol and Progol+ in order to extract information from DIPs. However, as shown in the figure, alongside MLW, there is a module called *Pe-Pl*. We discussed *Pe-Pl* in the background chapter (§2.9) and will be exploring the methodologies GH uses to harness *Pe-Pl* in the following section.

7.4 GH employing *Pe-Pl* and Reality Factor

Over the past few years, as explained in chapter 2 (§2.9), much research has been done focusing on using probability theory along with inductive logic programming and machine learning. For instance, *Stochastic Logic Programs* (SLP) [88, 87, 85] are an extension to ILP systems, providing a probabilistic logic representation in order to enhance the induction process while the background knowledge represents uncertainty. An SLP system is well suited for our DIP problems where it is not certain if the same relations learned for the profile of the guilty party between the suspects are held in the current case. Solving a DIP involves learning a set of hypotheses from the case studies in order to be used in the current case. For instance, Weka and Progol, both yield conjectures about the case studies or in particular the “guilty” suspect(s). GH only extracts the conjectures which cover 100 % of the positive examples and none of the negative examples. Nonetheless, we should bear in mind that these conjectures might not cover all the examples of current case. In other words, the current case examples can have different degrees of certainty compared to their corresponding examples in the past cases. Therefore, SLP as a framework which provides probabilistic learning along with inductive logic programming (PILP) [46, 105], is a useful tool to prioritize the generated hypothesis from Progol, Weka or any other module that can be employed in the GH machine learning workspace.

In this section, we define how GH has employed SLP in order to prioritize the hypotheses generated in GH’s MLW. SLP can consist of a set of definite clauses, each with a probability tag. In some forms of SLP programs, some of the clauses may not have any probability label. An example of an SLP from [40] is shown in figure 7.11. The figure depicts that the sum of probability labels for of each definition is 1. In some cases, known as *incomplete* SLPs, the total is less than 1.

```

0.4 :: s(X):- p(X), p(X).
0.6 :: s(X):- q(X).
0.3 :: p(a).
0.7 :: p(a).
0.2 :: q(b).
0.8 :: q(b).

```

Figure 7.11: An example of an SLP adapted from [40]

With the use of the *Failure-Adjusted Maximization* (FAM) algorithm explained in [40], SLP learns the parameters of each of the definite clauses and outputs their maximum likelihood estimates. GH uses the *Pe-Pl* software [27], which has an implementation of the FAM algorithm for SLP learning and was briefly explained in (§2.9). As shown in figure 7.11, clauses in probabilistic logic programming are annotated with probability values. Let P be the probability distribution that covers the set of examples E , hypotheses H and the background knowledge B : $P(E | H, B)$. With the use of *Pe-Pl*, GH iteratively learns the hypothesis H^* that has the maximum likelihood in covering the current case. The problem is to select the highest likelihood hypothesis among all the hypotheses generated by MLW in the GH solution module. In order to prepare the data for *Pe-Pl*, it is needed to define the background, set of examples and set of hypotheses. H is the set of conjectures generated by MLW where we initially assign equal probability distribution among all of them in the *Pe-Pl* system. The equal probability distribution is due to the fact that we assume all the extracted conjectures from MLW to cover all the positive and none of the negative examples provided in the case studies. With the use of *Pe-Pl*, we want to investigate which of the conjectures are more likely to explain the current case.

The background knowledge B , provided to *Pe-Pl*, is the same as the current case clauses. The only tweak needed in order prepare our DIPs for *Pe-Pl*, is to provide a set of examples with probabilistic labels. Considering that the aim is to find the most probable hypothesis explaining the current case, it is not possible to use the examples given in the case studies; as explained earlier those examples express the relationships between the suspects and predicates in the past cases where those relationships are not necessary the same in the current case. Therefore, we need to have a kind of estimation over the suspects in the current case fulfilling the profile of the guilty party. This can be done through the heuristics mentioned as the Reality Factor in chapter 5. In §(5.2.2), we used a set of weighting functions in order to have an initial estimate of guilty suspects in the current case so as to generate more realistic past cases. We also mentioned this methodology can be evaluated as a value-laden, self solution method without using any of the constraint solving or machine learning techniques. The weighting functions can rank the most significant predicates and suspects generated randomly in the current case. Recalling from

chapter 5, table 5.8 illustrated the hierarchy of the set of suspects in an example of a current case. A part of that table is as follows:

Rank: {d1=12, a1=6, f1=6, a2=6, e2=6, d2=6, e1=0, b1=0}

The table shows the suspects associated with a weight indicating the possibility of being the answer. These weights are normalized and converted to the likelihood degree labels to be fed to *Pe-Pl*. Hence, the background knowledge, the set of examples and hypotheses along with their probability labels are fed to *Pe-Pl* which aims to select the most probable hypothesis. An example of the input to parameter estimation software is shown in figure 7.12.

```
% Rules
0.100 :: murderer(A):- pred1(A).
0.100 :: murderer(A):- pred4(A,_,_), pred8(A). % best hypothesis
%Current Case Scenario
% Examples
0.400 :: murderer(e2).
0.300 :: murderer(d1).
0.200 :: murderer(a1).
0.100 :: murderer(f1).
% Background
pred1(b2).pred1(a1).pred1(d1).pred2(e2,d2).
pred2(d1,a1).pred3(f1,d1).pred3(d1,e2).
pred3(f2,e1).pred3(a1,e2).pred3(a1,b2).
pred3(d1,d2).pred3(b1,e1).pred4(f1,d1,e1).
pred4(f1,d1,d2).pred4(e2,b2,a1).
pred4(e2,d2,b2).pred4(e2,d1,f2).
pred4(b2,b2,d1).pred5(d2).
pred5(f2).pred6(a2,d1).
pred7(e2,b2).pred7(a2,a1).
pred8(f1).pred8(b2).pred8(e2).
pred9(b1,d1,f2).pred9(b2,e1,b1).
pred10(f2,a1,b2).pred10(b2,e2,a1).
```

Figure 7.12: Input to *Pe-Pl*

We will discuss the configuration and settings of *Pe-Pl* suite in chapter 8, but suffice to say, the parameter estimation is achieved iteratively using the algorithm mentioned in [105]. Figure 7.13 shows the result of the probability estimation for the rules mentioned in figure 7.12. It is clear that after 10 iterations, (the number of iterations is set manually), the second rule: `murderer(A):- pred4(A,_,_), pred8(A)` is correctly associated with a higher probability (0.52). The most possible hypotheses are consequently fed to *solver* modules.

```

results
:

log_likelihood(-23.5375753890101)
Iteration(10).
  1:0.47058823529411764053,  2:0.52941176470588235947

```

Figure 7.13: Correctly selected Hypothesis

In this section, we showed that using a combination of machine learning and *Pe-Pl* systems, GH can choose the best among all the hypotheses generated by various machine learning tools. We briefly discussed the details of *Pe-Pl* and parameter estimation in chapter 2 (§2.9). Fundamental details of parameter estimation and probabilistic inductive logic programming is beyond the scope of this thesis, nevertheless, it is important to mention that this approach is still at its early stages. According to some of its pioneers, the authors of [105], more work is needed to make PILP as efficient as traditional inductive logic programming systems. However, the promising results of its primary analysis in this project can hopefully inspire further research and application in DIPs.

7.4.1 Partial Solver

A Partial Solver for Cluedo scenarios was described in chapter 6 (§6.3.2). The same methodology is applied for generic DIPs. The only difference between *MLW output* and *Probabilistic output* mentioned in figure 7.1, is the number of answers. In MLW output, GH shows the answers associated with every hypotheses extracted from the machine learning workspace while the probabilistic output shows only the answer related to the best hypothesis picked from the set of extracted rules. An example of the partial solver output was shown in chapter 6, table 6.2. Further details of the answers and analysis of the results will be shown in chapter 9.

7.5 Conclusions

In this chapter, we initially showed our design of GH's solution module and described how GH employs various techniques in concert in order to solve a DIP. We discussed the use of combined reasoning techniques and showed how the machine learning workspace, comprised of various tools, can be employed along with different solvers to tackle a problem that cannot be solved by any of the standalone systems. We also showed that the power of the combined system draws upon the various systems embodied in GH and can be more than the sum of their individual constituents. We showed how different versions of the same reasoning approaches such as Weka, Progol and Progol+ and also disparate approaches such as HR and CLPFD can be used to fit our purpose. A combination of probabilistic and inductive logic programming approach was also described as a way to choose the most probable hypotheses describing the current case of a DIP. In addition, we showed how different solvers can enable GH in solving different representations of a problem.

Chapter 8

Experimentations and Configurations

8.1 Introduction

One way to demonstrate that the GH system performs well in solving DIPs is to illustrate its performance on a large set of problems. In §7.3 we described what we mean by a solution and in this chapter we discuss how to evaluate GH's solutions based on the systems it employs. In §8.2, we recall the formulae for a DIP solution and discuss how we calculate its error rate. GH has been designed to generate various type of DIPs in terms of difficulty and we will investigate the differences between these types in §8.3. We will also discuss the usefulness of the reality factors introduced in chapter 5. In this chapter, we will explain a set of experiments that have been designed to evaluate GH's performance. Those experiments will provide a platform to compare various modules of GH, mainly, with respect to their efficiency.

In chapter 4, we discussed how DIPs are dynamic in the sense that they change at different time steps. Each time step will bring to light further information about the current case that can result in different likelihood degrees for the suspects. In §8.4 we investigate the experiments in which GH generates DIPs partially at each time step and finds the respective temporal solution(s). In §8.6, we show that the execution time, in addition to error rate, is another evaluation factor which influences our choices of GH's encompassed systems. As such, error rate and execution time are both our yardsticks for comparing the GH embodied systems. We

will discuss the results of such comparison in chapter 9. In this chapter, we will also demonstrate a worked example in which we compare Weka, HR and Progol.

Various modules incorporated within GH interact through a pipeline of outputs resulting from a subset of systems and fed to another subset and vice versa. The entire operation requires a large amount of configurations and implementations in order to guarantee that the bespoke systems can smoothly interact and are flexible to various input/outputs. Some of the configuration and implementation details of the GH framework are explained in §8.7.

8.2 Calculating the Error Rate of a Solution

Recalling from section 7.3, the set of solutions (answers) produced by GH can be formalized as shown in figure 8.1. In each DIP, there are a number of possible diagnoses/suspects, and the problem is to rank them in order of increasing likelihood of being the cause of the illness/guilty of the crime (which we call the *target* candidate). Therefore, the rank of each suspect indicates how likely that suspect is to be the target candidate. One obvious way to do the ranking is in terms of how many constraints are upheld by each candidate.

$A = \{a_1, \dots, a_n\}$ is a set of solutions where:
 $a_i = \{S'_i, \dots, S'_m\}$ where m is the *sizeof*(*SuspectDomain*)
 S'_i is the likelihood degree of Suspect S_i where $S_i \in \text{SuspectDomain}$

Figure 8.1: ‘A’ representing the set of solutions

We mentioned in chapter 4 that for each generated DIP scenario, the correct embedded answer is stored in order to be used later for the evaluation of the GH’s solutions. The stored answers are compared with solutions to calculate the error rate. There are various suggestions about how to perform the error analysis and measure the accuracy of a forecast or prediction with regard to the eventual outcomes. Mean squared error (MSE), mean absolute percentage error (MAPE), root mean square error (RMSE), mean absolute error (MAE) and median absolute error (MdAE) are some commonly used methods for comparing an estimate to an exact value

100% d2	100% d2
50 % a1	66 % e1
50 % b1	66 % e2
50 % e1	33 % a1
50 % e2	33 % b1
50 % f1	33 % b2
25 % a2	33 % d1
25 % b2	33 % f1
25 % d1	33 % f2
25 % f2	

(a) embedded answer (b) solution: 12 % error rate

Figure 8.2: Comparing the GH solution with the embedded answer.

[62]. Deciding on which method is the most applicable for a particular dataset, is an area of contention for many researchers. However, most of these methods are chosen depending on the nature of the dataset. In our analysis, we need to measure the difference between GH's solutions and the exact answer embedded in each of the DIP scenarios. The percentage next each suspect indicates the rank of each suspect. In order to better understand what we mean by the difference between the solution and the correct answer, an example is shown in figure 8.2. Figure 8.2a depicts the embedded answer in a DIP scenario whereas GH's obtained solution is illustrated in figure 8.2b. The error rate is the difference between the likelihood degrees obtained from GH and the corresponding values in the embedded answer.

Root mean squared error and mean squared error are both popular methodologies for calculating the error rate. Their only drawback is the higher sensitivity to outliers compared to methods such as mean absolute error (MAE) which measures the average magnitude of the errors. In other words, they assign higher weighting to data which is far outside the norm and could be more useful when we want to particularly highlight the unusual events, i.e., when large errors are particularly undesirable [54]. In order to calculate the individual differences of a suspect, we calculate how much each suspect's likelihood is off, or mistaken by, from the actual value. For instance, as shown in figure 8.2, **a1** is 50% likely to be the guilty suspect whereas its outcome likelihood is 33%. There is no reason to weight any of the individual differences unequally, therefore, we chose mean absolute error (MAE) for calculating the error rate according to formula 8.1. In this formula, n is the total number of suspects, F'_i is the likelihood degree of

$$MAE = \frac{1}{n} \sum_{i=1}^n |F'_i - S'_i| \quad (8.1)$$

the suspect S_i calculated by GH and the S'_i , as mentioned earlier in figure 8.1, is the actual likelihood degree of S_i . Based on this formula, the error rate of the GH solution shown in figure 8.2b is 13.2%. Note that the suspect **a2** is missing in the GH's solution which means that GH has assigned the value 0 for it. Therefore, the difference is assigned as 25%. Note that GH may conclude that a suspect is a bystander and its likelihood degree is 0%, hence the suspect will not appear in the solution list.

8.3 DIP types

Real-mode DIPs, as explained in chapter 4, page 78, are classified based on their levels of difficulty. Different levels of difficulty are formed with respect to (a) the number of rules embedded in the current and past cases, (b) the number of suspects that satisfy all the rules and (c) the methods by which the case studies are generated. To recapitulate, case studies are either generated similar to the current case where the only difference is the removal of some of the predicates and suspects in the case studies or they are generated based on the heuristics explained as reality factors in chapter 5.

DIP types in terms of their difficulty levels are as follows:

- *Type-1*: There are multiple rules embedded in the current and past cases.
- *Type-2*: There is only one single rule embedded in the current and past cases.
- *Type-3*: Case studies are generated while taking the reality factors into account. There could be single or multiple rules embedded in each scenario.

Recalling from the previous chapter, formula 7.2, the GH problem generator has been designed in such a way that it can embed a set of constraints, each as a single clause with a given number

$$\begin{aligned}
&\text{Let } C = \{C_1, \dots, C_k, \dots, C_m\} \text{ be the set of rules} \\
&\text{Let } C_k \text{ be the most specific (MS) rule in } C \\
&\forall 1 \leq i \leq m, P_i \in \{P_1, \dots, P_m\} \text{ such that} \\
&P_i \text{ is the set of predicates appearing in the body of } C_i \\
&\forall P_i \in \{P_1, \dots, P_m\} \longrightarrow P_i \subseteq P_k
\end{aligned} \tag{8.2}$$

Figure 8.3: Attribute of the ruleset

of literals L , conjoined in the clause, and a maximum arity A for the literals. Therefore, the pair (LA) , elaborates the specification of embedding rules in each DIP.

8.3.1 Level of difficulty: Type 1

As discussed previously, every DIP consists of a current case and a set of case studies. The current case can be modeled as a constraint satisfaction problem where the constraints (rules) are imposed on a set of suspects. Often, however the constraints are not sufficient to determine the guilty party and a set of previously solved cases can be useful in the sense that the additional constraints can be learned from them and applied to the current case. Applying the learned constraints on the current case can hopefully enable GH to solve the problem.

There are multiple rules embedded in the case studies of a type-1 DIP, all of which correctly classify the guilty/not guilty suspects. Rules are generated and embedded based on the algorithms mentioned in §4.4. DIPs are designed to contain general or specific rulesets. Depending on which type of rulesets is embedded in DIPs, the solutions may change; we will elaborate this further in chapter 9 (§9.2.6) and will argue the usage of Occam's razor [11, 119] for certain DIP scenarios. We will also discuss that the most specific rules, at times, should be deemed better, i.e., discarding the Occam's advice.

The ruleset C holds the attributes shown in figure 8.3. According to the figure, each set of predicate appearing in the ruleset is a subset of the predicates appearing in the most specific (MS) rule. In other words, the set of predicates appearing in the MS is the union of all the predicates presented in the rules. A set of rules embedded in a DIP scenario is shown in figure

8.4. According to the figure, the maximum number of predicates involved in the rules is 5 and the maximum arity of predicates is 6.

8.3.2 Level of difficulty: Type 2

In type-2 DIPs, a single rule is embedded in the current and past cases. That is, only one rule can correctly classify the guilty/not guilty suspects. We will show that the performance of GH's bespoke systems, in terms of learning the rulesets, deteriorates compared to the former DIP type. The deterioration is manifested in the execution time and also the resultant ROC characteristics [51] of the learned rules. The results will be discussed in chapter 9, but it suffices to say that Weka, Progol and HR, spent longer times learning the rule compared to type-1; nonetheless, the sensitivity and specificity of the learned rules revealed that some suspects had been wrongly classified as guilty/not guilty (false positives and negatives).

According to the generation algorithms discussed in §4.4, only a subset of suspects are incorporated in the literals that are conjoined to form the constraint. We should note that incorporating a suspect is predicated on embedding the suspect in the $C_{Locations}$. Recalling from page

```

-----
All the rules
answer(A):- pred2(A), pred5(_X4,_X3,_X2,A,_X1).
answer(A):- pred7(_X5,_X4,_X3,A,_X2,_X1), pred8(A).
answer(A):- pred2(A), pred5(_X4,_X3,_X2,A,_X1), pred8(A).
answer(A):- pred1(_X4,_X3,_X2,_X1,A), pred2(A), pred5(_X8,_X7,_X6,A,_X5).
answer(A):- pred1(_X8,_X7,_X6,_X5,A), pred5(_X4,_X3,_X2,A,_X1), pred8(A).
answer(A):- pred1(_X8,_X7,_X6,_X5,A), pred2(A),pred5(_X4,_X3,_X2,A,_X1),
                                           pred8(A).
-----
Most General
answer(A):- pred2(A), pred5(_X4,_X3,_X2,A,_X1).
answer(A):- pred7(_X5,_X4,_X3,A,_X2,_X1) & pred8(A).
-----
Most Specific
answer(A):- pred1(_X4,_X3,_X2,_X1,A), pred2(A),
           pred5(_X4,_X3,_X2,A,_X1),pred7(_X5,_X4,_X3,A,_X2,_X1), pred8(A).

```

Figure 8.4: An example of an embedded ruleset – Type-1 DIP

84, $C_{Locations}$ specifies the main suspect's locations in each of the literals embedded in the rules. The algorithm can only produce type-1 DIPs, however, with a slight modification, GH can also generate type-2 DIPs.

Let us assume a rule containing k literals in the body where $k > 1$. Clearly, for $k = 1$, all the guilty suspect(s) and none of the bystanders should appear in the literals (by appearing we mean they should be presented in the $C_{Locations}$, otherwise they can appear in the other locations of the literals). In order to make sure that only a single rule is embodied in the DIP, for $k > 1$, GH should be restricted to establish *unique* intersections between k combination of predicates. As such, each combination should contain $k - 1$ predicates. The combinations are established with the not guilty suspects (negative examples). As an example, imagine the following rule is embodied in the ruleset:

$$\text{answer}(A) :- \text{pred1}(_, _, _, A), \text{pred2}(A), \text{pred5}(_, _, A, _).$$

Here, $P = \{\text{pred1}, \text{pred2}, \text{pred5}\}$ is the set of literals that appear in the body of the ruleset. The restrictions on the generation algorithm guarantee that all the rules, with the set of predicates P_k , where $P_k \subset P$, are also satisfied by the negative examples. Therefore, the number of combinations required for each rule with k predicates is: $\binom{k}{k-1} = {}^k C_{k-1} = \frac{k!}{(k-1)!(k-(k-1))!} = k$

8.3.3 Level of difficulty: Type 3

For this type of DIP, past cases are generated based on the reality factors, introduced and discussed in chapter 5. Reality factors are a set of formulae by which GH speculates about more/less important suspects and predicates presented in the current case. As a result, past cases are generated based on these rankings. We showed that by applying reality factors, GH is able to generate more realistic scenarios. Nonetheless, with the use of formulae for calculating ranks and weights, GH forms an initial speculation about the current case and can solve DIPs without the help of previously solved cases. Therefore, evaluating GH's solving performance for type-3 DIPs is not a test for the bespoke machine learning systems, rather, is an evaluation of GH's ability to solve a current case on its own. Let us consider a current case of a DIP with

the following rule and embedded answer:

```
answer(X) :- pred5(_,_,X),
              pred9(_,X), pred10(_,X).
100 % e1
66 % a1
66 % a2
66 % b1
```

GH, employing the reality factors, solves the problem where one of the answers can be found in figure 8.5 on page 173. As shown, **e1** is correctly highlighted as the most likely answer along with **b1** which in reality has 66% likelihood. The extracted rule has also correctly determined the important predicates in the problem description, however, **pred5** does not include the main guilty suspect. Generating the past cases based on reality factor is based on two major tasks: (a) conjecturing about the current case and extracting the suspects and predicates with higher weights and (b) spawning a set of past cases based on the speculations. We will show later in chapter 9 that in certain DIP scenarios in which the past cases are not generated based on reality factors, Weka, for instance, can learn the embedded rule(s) with 100% accuracy. Taking the reality factors into account, however, can change the solution accuracy. Evaluating the results in such cases can give us a better understanding of how “*well*” GH can solve the problems without using the machine learning workspace discussed in §7.2. As discussed previously in §8.2, assessing how good a solution is, can be achieved by performing the error rate calculations.

```
answer(X) :-
    pred5(Y, Z, A1),
    pred9(B1, X),
    pred10(C1, X).
100 % b1      33 % d1
100 % e1      33 % d2
66 % a1       33 % e2
66 % a2       33 % f1
33 % b2       33 % f2
```

Figure 8.5: GH result using the reality factors

8.4 Dynamic aspect and Temporal Solutions

As discussed in §3.5, DIPs are dynamic in the sense that they change at various time steps. As time passes, more evidence is gathered from the current case and more information comes to light from the past cases. GH can look for a solution at each time step, which we call a *temporal solution*, and ultimately, the final solution is obtained when no more information is fed to GH.

In our experiments, we evaluated GH's temporal solutions in order to gauge GH's performance at different time steps. For instance, calculating the average error rates when 20% of the information (suspects, facts and rules) is available. We should note that DIP types, as explained in page 170, are elaborated into constituent factors of the embedded rules: (a) number of literals conjoined to form the body of the ruleset(s) and (b) the maximum arity of those literals. In order to compare GH's solution performance, we needed to make sure that all DIPs are evaluated on a fair basis. Therefore, we designed GH to equally distribute the crucial information amongst time intervals. GH generates the current case in such a way that at each time step a part of the embedded rule(s) is revealed. In summary, the set of literals appearing in the rulesets are divided into equal subsets where each subset will be presented at each time step.

Recalling from the ruleset attributes shown in figure 8.3, $P_i = \{p_1, p_2, \dots, p_k\}$ is the set of predicates appearing in the body of the rule C_i . In our dynamic experiments, at each time step, GH reveals one of the predicates appearing in the P_i . Therefore, all the DIPs with rules that contain n predicates, are evaluated in n time intervals. Let us consider an example shown in figure 8.6. The figure depicts that the embedded rule contains predicates 4, 5 and 8. As such, these 3 predicates should appear in 3 time intervals. It is clear that **pred4** appears in $t = 1$ along with other facts containing **pred1**, **pred2** and **pred3**. Similarly, other predicates in the later time intervals. This way, 33% of the embedded rule is always provided to the GH solution module at each interval.

The schematic diagram, representing GH's mechanism for evaluating temporal solutions, is shown in figure 8.7 on page 176. As shown in the figure, a segment of the current case is generated at each time step and all the past cases are spawned according to that particular

```

% answer(X):- pred4(X), pred5(_,_ ,X), pred8(X).

% at Time = 1          % at Time = 2          % at Time = 3

pred1(e1,d1).          pred5(d1,e1,b1).          pred8(b1).
pred2(a1,f1).          pred5(e2,d2,a2).          pred8(a1).
pred3(a1,d2).          pred5(e2,d1,b2).          pred8(b2).
pred4(b1).             pred6(e1,f2,d2).          pred9(e1).
pred4(a1).             pred6(f2,d2,d1).          pred9(b2).
pred4(a2).             pred6(d1,a1,a2).          pred10(b2,d2,a2).
                      pred7(a1).          pred10(f2,a2,f1).
                      pred7(f2).          pred10(a2,e1,d2).
                      pred7(a2).

```

Figure 8.6: Distribution of information in dynamic experiments

segment. Therefore, each DIP at $t = k$, only contains the information obtained at $t = k$ from the current case and past cases. DIPs (current and past cases) are fed to Weka, Progol and HR as part of the solution module described in chapter 7 (§7.2). However, we eliminated HR in the figure for easier illustration. The error rate is calculated at each time interval. Bear in mind, that we calculate the error rate in accumulated time intervals. For instance, let us consider a case where the problem is distributed in three time steps as shown in figure 8.6. In order to calculate the error rate when 66% of the information is provided, we need to consider the error rate when $t = \{1, 2\}$ or $t = \{1, 3\}$ or $t = \{2, 3\}$. The error rate is achieved by averaging over the error rate of all the possible combinations of time intervals.

8.5 Experiments

Experiments are represented with a combination of L and A , where L indicates the maximum number of literals in the embedded ruleset(s) and A represents the maximum arity of those literals. For instance, $L3A6$, indicates that there are 3 predicates in the embedded rule with maximum arity of 6. Error rates are achieved by averaging over the errors of 1000 generated DIPs. For every level of difficulty, discussed in the former section, and for every combination of L and A , we generated 1000 DIPs, randomly, in order to have a confident error rate evaluation. In addition, for each DIP type, there is either one or more guilty suspects. GH stores the error

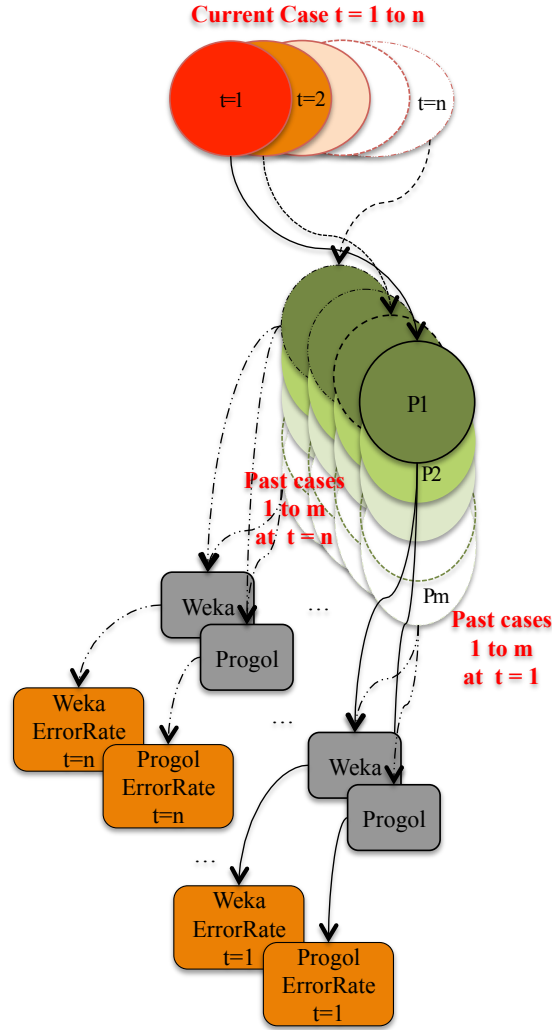


Figure 8.7: An overview of dynamic process (generation and solution) in GH

rates of each experiments in a file divided into a header and a body. The header contains the experimentation details and the body stores the rules obtained from the employed machine learning systems along with the error rates obtained from applying those rules. An example of a header can be found in figure 8.8.

Table 8.1 shown in pages 178 and 179, illustrates the details of the experiments. On page 178, the table depicts the first three types of DIPs that we experimented with using Weka, Progol, HR2 and HR3. The last system embodied in GH is HR3 which is an updated version of HR2 and discussed briefly in §2.6.5. The summary of the DIP specifications is mentioned at the end

DIP type(Complexity)	TYPE-1
Guilty Suspect(s)	>1
Dynamic is	OFF
A: Max Arity	1
L: No of Literals	3
Number of Runs	1000

Figure 8.8: The header of a file that stores the error rates

of each section of the table. Bear in mind that GH generated two versions of type-1 and type-2 DIPs based on the number of embedded guilty suspects; one version contains only one suspect that satisfies all the constraints whereas the other version holds more than one suspect. In all the generated DIPs, there are other suspects that satisfy only a few of the constraints, hence they have lower likelihood of being the cause of the illness/guilty of the crime.

In addition, we have investigated the error rates in dynamic mode where a segment of a DIP is produced and fed into the solution module. Ultimately, we experimented with Weka's various measurements of interestingness (confidence, conviction, leverage and lift), explained in chapter 2 (§2.7.1). We compared the performances of Weka's various metrics in type-1 (with multiple rules embedded) and type-3 (generation based on reality factors), in order to investigate which of the criteria is the most appropriate for sorting the resulting rules in different DIP types.

No	Level of difficulty	Commentary
1	Type-1	<u>LxAy Combinations:</u> L3A3, L3A4, L3A5, L3A6, L4A3, L4A4, L4A5, L4A6, L5A3, L5A4, L5A5, L5A6, L6A3, L6A4, L6A5, L6A6 <i>Number of generations for each LxAy: 1000</i> <i>Systems Compared: Weka, Progol, HR2, HR3</i> <u>Spec:</u> The ruleset in this level of difficulty should satisfy the conditions shown in figure 8.4. Two sets of experiments with number of <i>target candidates</i> = 1 and > 1.
2	Type-2	<u>LxAy Combinations:</u> L3A1, L3A2, L3A3, L3A4, L3A5, L3A6, L4A1, L4A2, L4A3, L4A4, L4A5, L4A6, L5A1, L5A2, L5A3, L5A4, L5A5, L5A6, L6A1, L6A2, L6A3, L6A4, L6A5, L6A6, L6A7, L7A1, L7A2, L7A3, L7A4, L7A5, L7A6, L7A7 <i>Number of generations for each LxAy: 1000</i> <i>Systems Compared: Weka, Progol, HR2, HR3</i> <u>Spec:</u> There is only one rule embedded in this type as discussed in §8.3.2. Two sets of experiments with number of <i>target candidates</i> = 1 and > 1.
3	Type-3	<u>LxAy Combinations:</u> L3A3, L3A4, L3A5, L3A6, L4A3, L4A4, L4A5, L4A6, L5A3, L5A4, L5A5, L5A6, L6A3, L6A4, L6A5, L6A6 <i>Number of generations for each LxAy: 1000</i> <i>Systems Compared: Weka, Progol</i> <u>Spec:</u> The generation of past cases is based on the reality factors as discussed in §8.3.3. There are multiple rules embedded in each current case.
continued on next page ...		

No	Level of difficulty	Commentary
4	Dynamic	<u>LxAy Combinations:</u> L3A1, L3A2, L3A3, L3A4, L3A5, L3A6, L4A1, L4A2, L4A3, L4A4, L4A5, L4A6, L5A1, L5A2, L5A3, L5A4, L5A5, L5A6, L6A1, L6A2, L6A3, L6A4, L6A5, L6A6 <i>Number of generations for each LxAy: 1000</i> <i>Systems Compared: Weka, Progol</i> <u>Spec:</u> The error rate is calculated for the temporal and final solutions as discussed in §8.4. There are multiple rules embedded in each current case.
5	Type-1,2: Weka	<u>LxAy Combinations:</u> L1A1, L1A2, L1A3, L1A4, L1A5, L1A6, L2A1, L2A2, L2A3, L2A4, L2A5, L2A6, L3A1, L3A2, L3A3, L3A4, L3A5, L3A6, L4A1, L4A2, L4A3, L4A4, L4A5, L4A6, L5A1, L5A2, L5A3, L5A4, L5A5, L5A6, L6A1, L6A2, L6A3, L6A4, L6A5, L6A6 <i>Number of generations for each LxAy: 1000</i> <i>Comparing Weka parameters: Confidence, Conviction, Leverage, Lift</i> <u>Spec:</u> In this set of experiments, we calculate the error rate for various Weka parameters explained in chapter 2 (§2.7.1). The generated DIPs are of type-1.
6	Type-3: Weka	<u>LxAy Combinations:</u> L3A1, L3A2, L3A3, L3A4, L3A5 L4A1, L4A2, L4A3, L4A4, L4A5 L5A1, L5A2, L5A3, L5A4, L5A5 L6A1, L6A2, L6A3, L6A4, L6A5 <i>Number of generations for each LxAy: 1000</i> <i>Comparing Weka parameters: Confidence, Conviction, Leverage, Lift</i> <u>Spec:</u> In this set of experiments, we calculate the error rate for various Weka parameters explained in chapter 2 (§2.7.1). The generated DIPs are of type-3 (based on reality factors).

Table 8.1: Summary of experiments

8.6 An example of the results

In this section, we will show partial segments of various results obtained from a single experiment on type-2 DIPs. In order to evaluate the answers, as discussed earlier in §8.2, GH is provided with the correct answer including the exact likelihood degree of suspects. According to figure 8.9, RUN TIME N reveals the result of the N^{th} generated DIP. At each run time: (a) the ruleset learned by each system, (b) its resulting error rate and (c) the likelihood degree of the suspects, are stored. Each bracket contains a decimal number that indicates the error rate obtained from applying the rule to the current case, followed by the ruleset and finally the suspects along with their likelihood degree. It is clear that at run time 1, all the systems have extracted the correct rule that yields the answer with zero error rate. However, at DIP number 5 and 997, HR cannot learn the correct rule. HR cannot learn the embedded rule, to any extent, at run time 685.

```
DIP type(Complexity)      TYPE-2
Guilty Suspect(s)        =1
Dynamic is                OFF
A: Max Arity              2
C: No of Literals         3
Number of Runs            1000

RUN TIME 1
Weka, Progol and HR2 are all equally good
weka   : [0.0, answer(X):- pred1(X, Y),pred5(X, Z),pred9(X)., {B2=100, A1=66, A2=66, B1=66}]
progol: [0.0, answer(X):- pred1(X, Y),pred5(X, Z),pred9(X)., {B2=100, A1=66, A2=66, B1=66}]
HR2    : [0.0, answer(X):- pred1(X, Y),pred5(X,Y),pred9(X)., {B2=100, A1=66, A2=66, B1=66}]
....

RUN TIME 5
Weka and Progol are equally good
weka   : [0.0, answer(X):- pred1(X, Y),pred8(X),pred10(Z, X)., {A2=100, A1=66, B1=66, B2=66}]
progol: [0.0, answer(X):- pred1(X, Y),pred8(X),pred10(Z, X)., {A2=100, A1=66, B1=66, B2=66}]
HR2    : [8.25,answer(X):- pred1(X, Y),pred8(X),pred10(Y, X)., {A2=100, A1=66, B2=66, B1=33}]
....

RUN TIME 685
Weka and Progol are equally good
weka   : [0.0, answer(X):- pred1(Y, X),pred2(Z, X),pred4(X)., {A2=100, A1=66, B1=66, B2=66}]
progol: [0.0, answer(X):- pred1(Y, X),pred2(Z, X),pred4(X)., {A2=100, A1=66, B1=66, B2=66}]
HR2    : []
....

RUN TIME 997
Weka and Progol are equally good
weka   : [0.0, answer(X):- pred4(X),pred6(Y, X),pred9(X, Z)., {F2=100, A1=66, A2=66, B1=66}]
progol: [0.0, answer(X):- pred4(X),pred6(Y, X),pred9(X, Z)., {F2=100, A1=66, A2=66, B1=66}]
HR2    : [26.5,answer(X):- pred4(X),pred9(X, Y),pred4(Y).,
          {A1=66,B1=66, F2=66, A2=33, B2=33, D1=33, D2=33, E1=33, E2=33, F1=33}]
```

Figure 8.9: Part of the storage file

In every experiment, GH also stores the summary of the results in a separate file. Figure 8.10 is an example of a summary file. It shows that Weka and Progol have mined a rule in all 1000 problems, hence 1000 answers. However, HR2 learned a rule in only 626 problems. The accepted answers indicates the number of correct answers produced by each of the systems. Weka shows the best performance while Progol shows slightly less satisfying results compared to Weka. The execution times and error rates, averaged over 1000 DIPs, are shown in the subsequent sections of the summary file.

```

Provided Answers
Number of Answers provided by Weka   : 1000
Number of Answers provided by Progol: 1000
Number of Answers provided by HR     : 626

Accepted (Correct) Answers
Weka      1000
Progol    989
HR2       306

Average time taken in seconds
WEKA      0.05
PROGOL    0.33
HR2       14.49

Average Error Rates
Weka      0.0%
Progol    0.26%
HR2       43.27%
```

Figure 8.10: Part of the results summary

8.7 Configurations and Implementation Details

As discussed throughout this thesis, GH embodies various techniques and systems in order to generate, translate and solve DIPs. The process by which GH generates problems, reformulates and translate them into various syntaxes and also combines various reasoning techniques to solve them, overall, is a linear process. The interaction between various modules of GH is a hard coded chain of data processing module, feeding the output of one system to another. GH is implemented mainly in Java and handles the overall control. It handles the calls to Prolog, Progol, HR, Weka, *Pe-Pl*, the translation suite and the CSP re-formulator. Each of

these systems require a set of configurations that are parameterized within GH, every time a DIP is generated. The drawback of GH is that it requires storing and operating on many files. The nature of the storage files can be categorized into: (a) the files that contain the problem descriptions; such files are translated and input to the embedded systems (b) the translated files in various syntaxes, (c) the files that contain the output of the embedded systems (d) configuration files for GH and its embedded systems and (e) batch files for running different modules. In addition, GH benefits from a set of useful shell commands such as (`awk`, `sed`, `tail`, `head`, etc) that help smooth the translation and reformulation of various outputs.

GH has a graphical user interface shown in chapter 4 (§4.6). The GUI has a configuration that enables the user to adjust the system to the required settings. The GUI is mainly used for demonstrations and single DIP operations; for instance, checking the temporal solution at each stage and investigating the change in likelihood degrees after addition or removal of one or more of the facts and/or rules. “`GH.jar`” aggregates all the classes of GH and is used to run GH from the command line without invoking the GUI. This way, we only bundle the configuration file with the `jar` files and various blocks of codes written in Prolog and shell scripts.

8.8 Conclusions

In this chapter, we discussed how to evaluate GH’s proposed solution in order to demonstrate the usefulness of GH as a combined reasoning system for solving DIPs. We mentioned the benefits of generating thousands of problems and explained the methodologies for calculating the error rate of the solutions. In addition, we demonstrated the details of DIP types in terms of their levels of difficulty. We also discussed the methodology by which we demonstrate and evaluate temporal solutions provided by GH.

DIP types provide us with a better understanding on how the embedded systems perform at different levels of difficulty. In table 8.1, we showed the designed experiments by which we will gauge GH’s performance in the next chapter. We discussed the configuration and implementation details of GH and demonstrated the methods by which GH has produced the results.

Chapter 9

Results and Analysis

9.1 Introduction

In the previous several chapters, we formally defined dynamic investigation problems. We introduced methodologies for generating and translating them into various syntaxes. Subsequently, we demonstrated techniques by which the bespoke systems were employed to solve the problems. In chapter 8, we discussed experiments on different DIP types and explained the implementation details for establishing an interaction between various modules incorporated within GH through a hardcoded pipeline.

In this chapter, we will discuss the results of the experiments shown in table 8.1 on pages 178 and 179. The results will highlight the performance of various systems such as Weka, Progol and HR on DIPs. Recalling from §8.2, solving a DIP entails distinguishing the *target* candidate(s) and ranking the suspects with respect to their likelihood degrees. GH performance is achieved by calculating the error rate of the proposed answers. By evaluating the results, we provide a comparison between the systems GH harnesses in terms of their efficiency in mining rules from the background data. In addition, enabling the “reality factors” will assess how efficiently GH can solve DIPs whilst employing its underpinning heuristics. We demonstrate how employing reality factors may lead to a highly efficient solving mechanism and improve the overall effectiveness of GH.

The law of parsimony or the rule of simplicity known as Occam’s Razor principle suggests that the simplest of theories, predicting the same results, is likely to be the best explanation in all experiments. Occam’s principle is followed by many machine learning algorithms. For instance, Progol chooses to maximize the information compression of inductive inference rules by incorporating “Occam compression” criteria [84]. In this chapter, we argue why GH, whilst solving DIPs, should discard the Occam’s advice and choose the most complicated hypothesis extracted from its embedded machine learning systems.

9.2 Experimental Results

Recalling mainly from chapters 3 and 7, dynamic investigation problems are designed to model, to some extent, a generic situation which may arise in some domains, say medical diagnosis or solving a crime. Solving DIPs entails using the facts about the problem coupled with prior knowledge about the domain to narrow down the candidates that are the cause of illness or guilty of the crime. Often, however, not all the essential information is readily available which leaves the problem with no outright solution. Some additional relevant information can be found in related past cases from which a profile of the guilty party can be learned and applied to the problem. GH, therefore, aims to mine relevant information from the past cases with the use of its incorporated machine learning systems. GH tackles DIPs by harnessing a combination of techniques explained in §7.3.

The difficulty of a DIP is predicated on the following conditions whereby the experiments have been designed: (a) the number of rules embedded in the current and past cases, (b) the number of suspects that satisfy all the rules and (c) the methods by which the case studies are generated. We should note that the embedded rules are represented in the form of $L_x A_y$ (with exactly x literals of arity at most y).

We also note that the results shown in this chapter are the outcome of hundreds of thousands of DIPs generated in total for various types and categories. Many of the results have been omitted as many of the experiments contributed only to the intermediate results by which we learned

System	L3A1	L3A2	L3A3	L3A4	L4A1	L4A2	L4A3	L4A4	L4A5	L5A1	L5A2	L5A3	L5A4	L5A5	Av.
Weka	20	30	30	30	30	30	30	30	40	30	30	40	40	40	32
HR3	60	90	130	170	70	110	160	200	260	120	170	230	300	430	179
Progol	70	100	160	280	70	100	190	400	1270	70	100	230	530	1960	395
HR2	2780	6160	31890	38480	2850	6520	35370	3990	48420	2960	6960	44520	51250	71640	27835

Table 9.1: Execution times in milliseconds for Weka, HR3, Progol and HR2, running on a 3.2GHz processor, averaged over 1000 Type-1 investigation problems requiring solutions of the form $LxAy$ (with exactly x literals of arity at most y)

how to make DIPs more difficult and how to tune the machine learning algorithms in order to improve their performance. For instance, the experiments shown in tables 9.1 and 9.2 have been executed 4 times; each time, Weka parameters were adjusted to perform better. In addition, HR3 was adjusted to do almost everything *on-demand*, contrary to the earlier versions of HR series where the majority of processing was done to no avail [36]. Therefore, the configurations in the previous chapter and the results in this chapter indicate the optimal parameters and performance of the systems.

9.2.1 Type-1 DIPs: Multiple Rules

GH generating module generates type-1 DIPs in which multiple rules are incorporated. Rules are produced based on the algorithms mentioned in §4.4. The solution may change depending on which subset of the rules is chosen, whereby we evaluate the performance of the machine learning systems explained. Table 9.1 depicts the execution time in milliseconds for Weka, Progol, HR2 and HR3. The time is averaged over 1000 problems requiring a solution in the form of L_xA_y . We see that Weka, HR3 and Progol are all very fast, taking less than a second per problem on average. HR3 performed the same amount of theory formation steps as HR2, but it was 155 times faster on average over all the problems comparing to HR2. This speed rises to 166 times faster for problem set L5A5. HR3 is more efficient compared to HR2 due to modifications in its design and search mechanism [36]. A new search reduction technique has enabled HR3 to avoid forming new concepts by conjoining old ones where each has less examples than the target concept. HR3, nonetheless, is still almost 6 times slower than Weka,

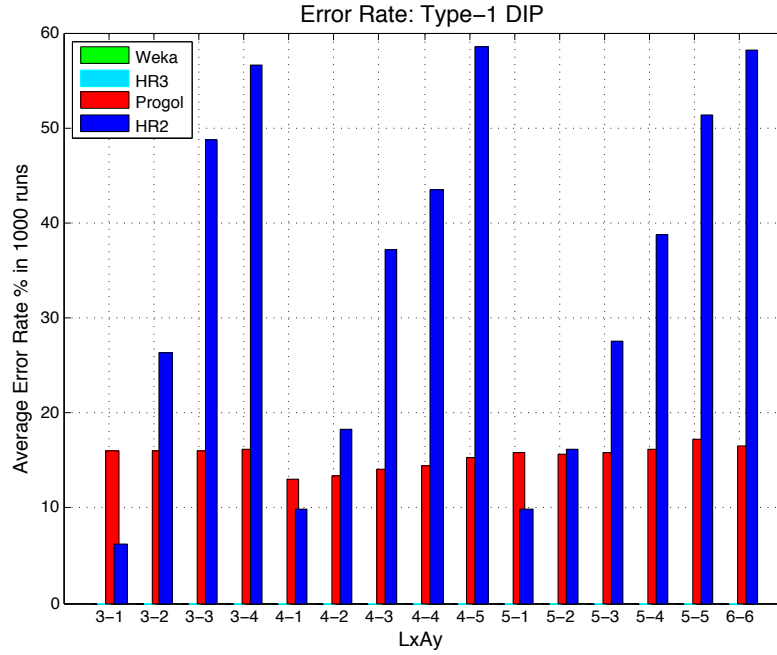


Figure 9.1: Error rates in type-1 DIP (Zero error rates for HR3 and Weka)

making it less efficient in mining the rules from investigation problems.

The error rates resulted from applying the extracted rules to the problem sets are shown in figure 9.1. As we see in the figure, Weka and HR3 both have zero error rates in all the problem sets. Weka, using the Apriori association rule mining algorithm, is configured with a confidence threshold of 0.99. As such, it will extract the rules with 99% or more degree of confidence as described in §2.7.1.

HR3 was able to expand the definition of equivalent concepts to completion and managed to find the correct embedded rules in all the problem sets. In many cases, the more general rules (the rules with fewer background concepts as their body literals), can correctly characterize the positive and negative examples in the background and can hence represent the profile of the guilty party. However, we will explain in §9.2.6, why the more specific rules should be considered in solving investigation problems by which the lower error rates can be achieved. HR3 is set to apply the existential production rule 10 times, as the arity of the background predicates can go up to 8 or 9. In addition, HR3 is also set to apply the conjunction production rule 4 times in order to search for more complex definitions and expand the definitions of the background concepts. This is so that the most specific solution can be found in each case. HR2, lacking

this ability, performed worse than HR3 with an average error rate of 32% as per the figure 9.1. Finding the most specific rule is not an easy task for HR2, given that it normally constructs the simplest concepts prior to the more complicated ones. HR2 can also come up with the more complicated rules at the expense of far more delay in the execution time, whereas, HR3 is clearly more efficient.

Progol uses *Occam Compression* which follows the Occam Razor's principle based on the information content to choose the most general inductive inference rules. Due to this, Progol cannot feasibly find the most specific rules. Progol's error rates, represented with red bars in figure 9.1, depict a value of 15%, averaged over all the problem sets. One may argue that the error rates do not imply the weak performance of Progol compared to the rest of the systems as it is programmed to find the simplest rules first. In section §9.2.2, we will investigate and compare the performance of Progol where there is only one single rule embedded in the problem description. In that case, the problem sets will provide us with a yardstick for a fair comparison. In addition, as promised earlier, we will explain in §9.2.6, why the most specific rules should be chosen in solving DIPs.

Another evaluation factor we should take into account is the ability of the employed systems to learn at least a single rule from the embedded ruleset. The learned rule, irrespective to how general or specific, can at least contribute to the result. Weka and HR3, were both able to do so in all the 1000 investigation problems generated in each L_xA_y category, while there were scenarios in which neither Progol nor HR2 could learn any of the rules in the ruleset. The performance declines in these two systems in terms of their ability to learn the embedded rules. For the problem sets shown in table 9.1, Progol was able to learn a rule in 98% of the cases whereas HR2 returned a solution in 76% of the problems; this HR2 percentage falls to 57.4% of the problems with solutions in form of $L5A5$.

According to table 8.1 in chapter 8, we designed another set of experiment in which DIPs contained more than one target candidate. HR2 was left out due to its low speed and high error rates shown earlier in this section. The results were the same for the problems requiring solutions with the target clause smaller than $L5A5$. However, the performance of the systems

System	L5A5	L5A6	L5A7	L6A5	L6A6	L6A7	L7A5	L7A6	L7A7
Weka	0	0	0	0	0	0	1.43	1.74	2.61
HR3	<1	<1	<1	<1	<1	<1	<1	<1	<1 (1550)
Progol	33	48	60	35	49	61 (4130)	35(5830)	49 (14040)	55 (92000)

Table 9.2: Error Rates for Weka, HR3 and Progol, running on 3.2GHz processor, averaged over 1000 Type-1 investigation problems (number of *target candidates* > 1) requiring solutions of the form LxAy. Significant execution times in milliseconds are given in brackets.

declined in the problems with higher LxAy. The results are shown in table 9.2. It is clear that the Progol error rate almost doubled in L5A5 compared to its previous rate (17%) shown in figure 9.1. Progol’s performance deteriorates with higher LxAy, nevertheless, it is clear that the maximum number of arity is more influential in the performance of Progol in comparison with the number of literals; with higher arity, Progol is less capable of learning the correct rule. So far, Progol could solve each problem in less than a second whereas the execution time rises in L6A6 and increases tremendously in L7A7 (7 times more than the slowest execution in the previous experiment).

In each category of LxAy, HR3 could not find the correct rule in almost 10 – 15 problems in a set of 1000 DIPs. Even though the error rate is still negligible, it is evident that HR3 is prone to error for this level of difficulty. The results reveal that HR3 outperforms association rule mining, in terms of error rate, when the number of literals is 7.

9.2.2 Type-2 DIPs: Single Rule

As explained in §8.3.2, for this type of DIP, we designed GH to incorporate relationships between the suspects in such a way that only a single rule could correctly classify the guilty/non-guilty suspects. That is, the correct solution entails the bespoke systems to construct the only rule which covers all the positive and negative examples. In this type of DIP, the fact that Progol only generates one answer (the simplest one) does not put Progol at a disadvantage as there is only one rule to be extracted. HR2 was excluded from the experimentations due to being slow in coming up with complicated solutions. Besides, such DIPs have been designed mostly

to cater for better evaluation of Progol in that they provide an equal ground for comparing Progol with the other systems.

Table 9.3 reveals the performance of Progol in terms of correct answers and error rates. Weka and HR3 could successfully solve all the problems. As shown in the table, Progol's performance declines with the increase in the number of literals and arity of the embedded rules. The numbers in the brackets show the ratio of correct rules to the total number of extracted rules. Clearly, both the numbers decrease with the increase of complexity.

We should note that Progol uses A* search over the hypotheses space. Thus, Progol guarantees to find the best hypothesis (with maximal compression) in case there exists a correct and complete hypothesis, i.e., the one that entails all the positives and none of the negative examples. However, Progol may exceed one of the resource parameters before forming any hypothesis, hence, Progol will be unsuccessfully terminated without learning the logic program implying the target literal. The results shown in the table was to evaluate Progol's performance while the parameters are set in such a way to avoid exhausting a complete search and try to find the solution in less than a second which is close to HR3 and Weka execution times.

In table 9.3 we present the average error rates, the number of generated solutions and also the number of correct solutions in total. However, we tried to have a better understanding in Progol's performance by finding the distribution of the error rates. As such, we need to know how reliable the estimated average error rate is. For instance, the average error rate in L4A3 is almost 9% which indicates that we can expect Progol to solve such problems with 9 percent error. Nonetheless, we need to find out how confident we can be in this number.

System	L3A1	L3A2	L3A3	L3A4	L3A5	L3A6	L3A7
Progol	0 ($\frac{1000}{1000}$)	0.46 ($\frac{979}{1000}$)	3.54 ($\frac{885}{989}$)	15.08 ($\frac{676}{914}$)	24.55 ($\frac{513}{844}$)	34.49 ($\frac{379}{726}$)	46.20 ($\frac{256}{656}$)
System	L4A1	L4A2	L4A3	L4A4	L4A5	L4A6	L4A7
Progol	0 ($\frac{1000}{1000}$)	0.69 ($\frac{986}{995}$)	9.03 ($\frac{842}{933}$)	25.74 ($\frac{606}{798}$)	48.30 ($\frac{351}{589}$)	55.41 ($\frac{230}{541}$)	64.93 ($\frac{153}{444}$)

Table 9.3: Percentage error rates of Progol averaged over 1000 Type-2 investigation problems requiring solutions of the form LxAy. The ratio of correct solutions to the total number of generated solutions is given in brackets. Progol parameters: i=5, nodes=1000

Figure 9.2 shows the Progol confidence intervals for various problem sets. The red lines in each bar indicates the mean value of all the error rates and the pink regions demarcate the 95% confidence interval for the mean. The blue region is the standard deviation. Each circle represents an instance of an error rate corresponding to a problem in each LxAy category. The figure is an improved representation of bar charts whereby we can also see how the data (in this case the error rates) is distributed. Presenting only the mean and the error bar can hide some aspects of the data behavior, however, overlaying the plot boxes by raw data, enables us to analyze its entire characteristics in a way that is not overwhelming.

We mentioned earlier that Progol solutions for L4A3 problem sets are expected to have 9% error rates whereas according to the figure we notice that the mean value of errors is roughly 3%; most of the errors lie on the zero value, hence the high density of black circles around zero. However, the charts show that the error rates can go up to approximately 50% at some

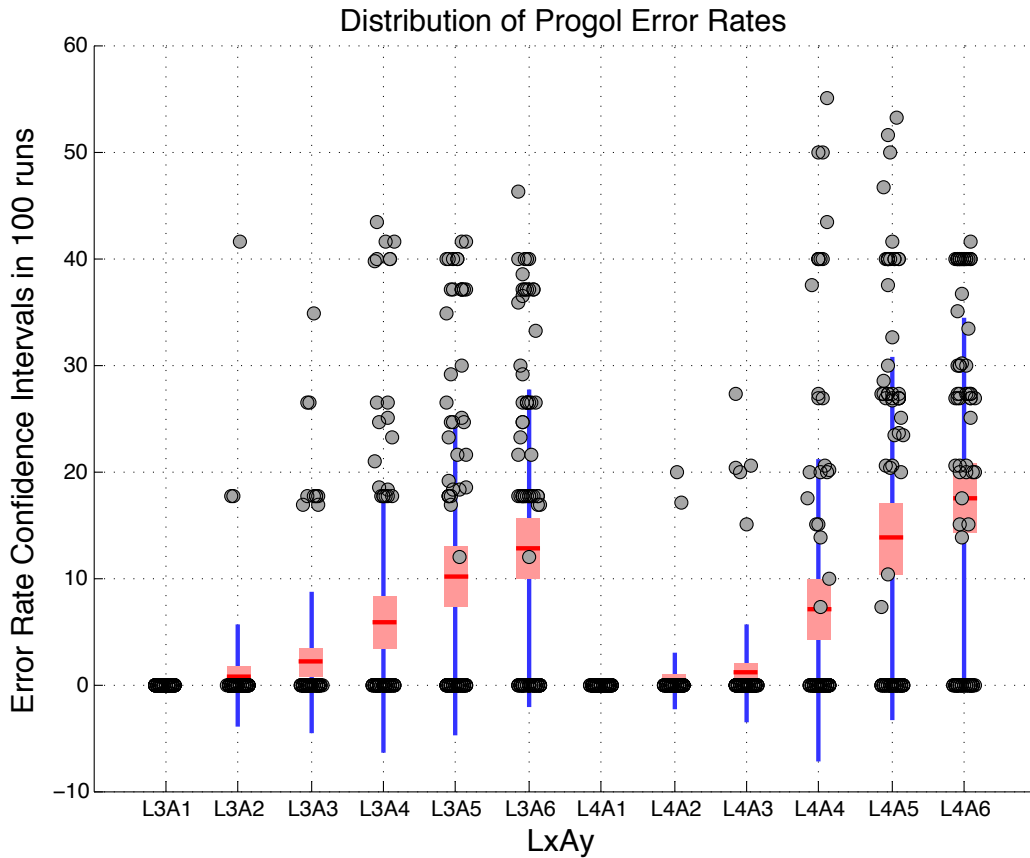


Figure 9.2: Distribution of Progol Error Rate - Type-2 DIPs

instances. Errors in L3A1 and L4A1 are all within the confidence intervals, covering the mean line, confirming that Progol is highly efficient in solving the corresponding category of problems. In the next section we will discuss the settings and possibilities in which Progol may achieve better results.

9.2.3 A note on Progol+

As discussed in §7.3.4, by using the mode declarations we describe a valid hypothesis in order to bias and delimit Progol search space. We built a wrapper around Progol, named Progol+, to test a combination of input/output prefixes in mode declarations whereby Progol can perform better. When specifying the body mode declarations for DIPs, we assign the predicate arguments with a ‘–’ prefix as we cannot be sure whether the argument should be instantiated before the predicate is called. Hence, we assign them as output to return a variable that may be used later as an input to another predicate. Assigning all the arguments as output will increase Progol’s search space tremendously, particularly when the arity of predicates rise to more than 3. Progol+ enabled us to use Progol in a more systematic way. By systematic, we mean, testing the performance of Progol rigorously in order to make sure the prefix used in the mode declaration produces the best possible combination in terms of results.

Progol+ revealed that for unary predicates it is best to assign the arguments as input. With this setting, Progol outperformed Weka that had proved to be the fastest in the previous experiments. As explained in §2.8.4, the maximum number of atoms in the body of any hypothesis clause, by default, is set to 4 in Progol. We needed to adjust the setting accordingly which is a drawback to Progol in that GH does not have access to information about the number of literals in the embedded rule. In summary, with Progol+ parameter setting, we realized that Progol can perform better than the other systems in the problem sets where the embedded rules contain only unary predicates. Table 9.4 provides a comparison between Progol, Weka and HR3 when the problems require solutions of the form LxA1. All systems perfectly solved the problems, however, as shown in the table, Progol performed faster.

When experimenting with Progol+ on the problem sets where the embedded rules had pred-

System	L3A1	L4A1	L5A1	L6A1	L7A1
Progol	<10	<10	<10	10	20
Weka	20	30	40	70	150
HR3	70	130	530	2150	14960

Table 9.4: Execution time in milliseconds, running on a 3.2GHz processor, averaged over 1000 Type-2 investigation problems requiring solutions of the form LxA1. Progol parameters: nodes=200, i=1, set(posonly).

icates with more than one argument, we realized that it would be best to assign one or more of the arguments as input in order to reduce the search space. We also needed to adjust the “nodes” and “i” parameters to hundred million and ten million, respectively, in order to complete the search in the problem sets where the number of arguments was more than 5. Increasing such parameters made Progol extremely slow compared to Weka and HR3, and highlighted the fact that Progol’s lattice is too large [113]. We briefly discussed lattices in §2.8.4.

Another issue with Progol in solving DIPs is the compression. As explained earlier, the learning goal of the ILP systems is to generalize from the provided examples and to build a theory that covers all the positive and none of the negative examples while having a good predictive power on the unseen examples. Progol uses Occam’s Razor as a good heuristic to achieve this, however, in order to build a theory with minimum description length, it uses a compression metric to evaluate the yielded theory. Amongst theories with the same predictive accuracy, Progol chooses the one with a higher compression measure. The compression is the weight of all positive examples covered, minus the weight of negative examples covered, minus the number of literals of the theory itself [113]. Note that GH typically generates 5 case studies for every current case (although this number can be easily modified), hence, the number of positive examples would be 5. Assuming that a theory provided by Progol does not cover any of the negative examples (best case), the weight of positive examples would be “5” and negative examples “0”. However, when the required theory is of the form LxAy, where $x > 5$, say 6, the compression would be $5 - 6 = -1$, i.e., there will be no positive compression. In other words, the theory is not compressive as it does not encode examples with less number of literals than the original theory; the original theory is the trivial lengthy one that covers all the positives and does not have any predictive power. Hence, Progol has difficulty dealing with

such scenarios. Besides, adding the positive examples (case studies) will make the problem less realistic. However, Progol+ revealed that “`set(posonly)`” which restricts Progol to learn only from positive examples and `set(inflate,10)` that multiplies all example weights by 10, will resolve this problem.

9.2.4 Type-3 DIPs: Reality Factors

Recalling from §8.3.3, in this type of investigation problem, GH forms an initial set of conjectures about the important pieces of information in the current case with the use of heuristics called reality factors, explained in chapter 5. The conjectures are deemed to rank the candidates with a consequent weighting of the predicates (facts) that describe the relationships between the candidates. The heuristics can be considered as a useful tool for solving investigation problems without relying on the case studies. Besides, the reality factors are utilized to generate more realistic scenarios. The error rates of the produced solutions can be used to assess how good the reality factors model the important information of the current case.

Weka, using association rule mining algorithm, suitable for finding the most specific hypothesis and Progol, utilizing Occam’s compression, were both employed in this experiments. The result of 12000 problems are shown in table 9.5. HR3 was excluded in this study as it was proved in the previous experiments to be as efficient as association rule mining in terms of finding the most specific hypothesis. According to the table, Progol’s performance is almost 4 times worse than Weka because Progol uses Occam Razor principle based on information content to choose the most general hypothesis. Hence, it returns solutions involving fewer background predicates than required. Given that Weka generated perfect solutions (with zero error rates) in the former

System	L4A3	L4A4	L4A5	L4A6	L5A3	L5A4	L5A5	L5A6	L6A3	L6A4	L6A5	L6A6	Av.
Weka	7(40)	8(50)	10(60)	10(60)	9(40)	10(60)	11(60)	12(70)	9(50)	10(60)	11(70)	12(80)	10(58)
Progol	34(270)	36(410)	39(530)	41(740)	35(290)	37(410)	39(540)	41(800)	36(300)	38(460)	40(650)	42(830)	38(519)

Table 9.5: Percentage error rates of Weka and Progol averaged over 1000 Type-3 investigation problems requiring solutions of the form $Lx Ay$. Execution times in milliseconds on a 3.2GHz processor are given in brackets.

types of DIPs, we can conclude that the “average” error rate of 10% implies the efficiency of reality factors, i.e., utilizing reality factors, GH can build a reasonable model of the current case.

9.2.5 Temporal Solutions

DIPs are comprised of a series of investigation problems which bring to light further information about the current case and can lead to further investigation focusing on the most likely candidates. In the previous experiments, we tried to illustrate the performance of GH in solving complete DIPs. By complete we mean that all the constituents of an investigation problem are available. In this section, we try to calculate the error rates when only segments of a DIP have come to light. This will enable us to gauge the temporal performance of GH’s solution mechanism. In §8.4, we showed how GH distributes the crucial information amongst time intervals in order to provide a fair basis for evaluating the embodied machine learning systems. As shown in figure 8.7 on page 176, the temporal solution at time $t = k$ is obtained by averaging over all the error rates that are achieved in k combinations of n , where n is the final instance whereby a DIP becomes complete. For instance, if a DIP is comprised on 4 investigation problems, all the essential information is fed to GH in 4 time intervals; calculating the error rates when $t = 2$ entails averaging the error rates of the solutions when 50% of the information is available, i.e., $\{IP_{\{1,2\}}, IP_{\{1,3\}}, IP_{\{1,4\}}, IP_{\{2,3\}}, IP_{\{2,4\}}, IP_{\{3,4\}}\}$. In summary, the GH task here is to focus on the extraction of additional constraints throughout an investigation. That is, while the constraints are not enough to narrow down the candidates to the target candidate directly, the extra constraints are used to rank them in terms of their likelihood of being the target. The results from the experiments are given in figure 9.3. The x-axis in each graph represents the time intervals that can be interpreted as available segments of DIPs in percentage, and the y-axis shows the corresponding error rates, averaged over 1000 DIPs. The graphs confirm that the larger number of constraints, achieved by more information fed to GH, would make the rankings more fine-grained than a smaller number. Hence, more information results in less error in both Weka and Progol. Nonetheless, figure 9.3 shows that in all the problem sets, Weka

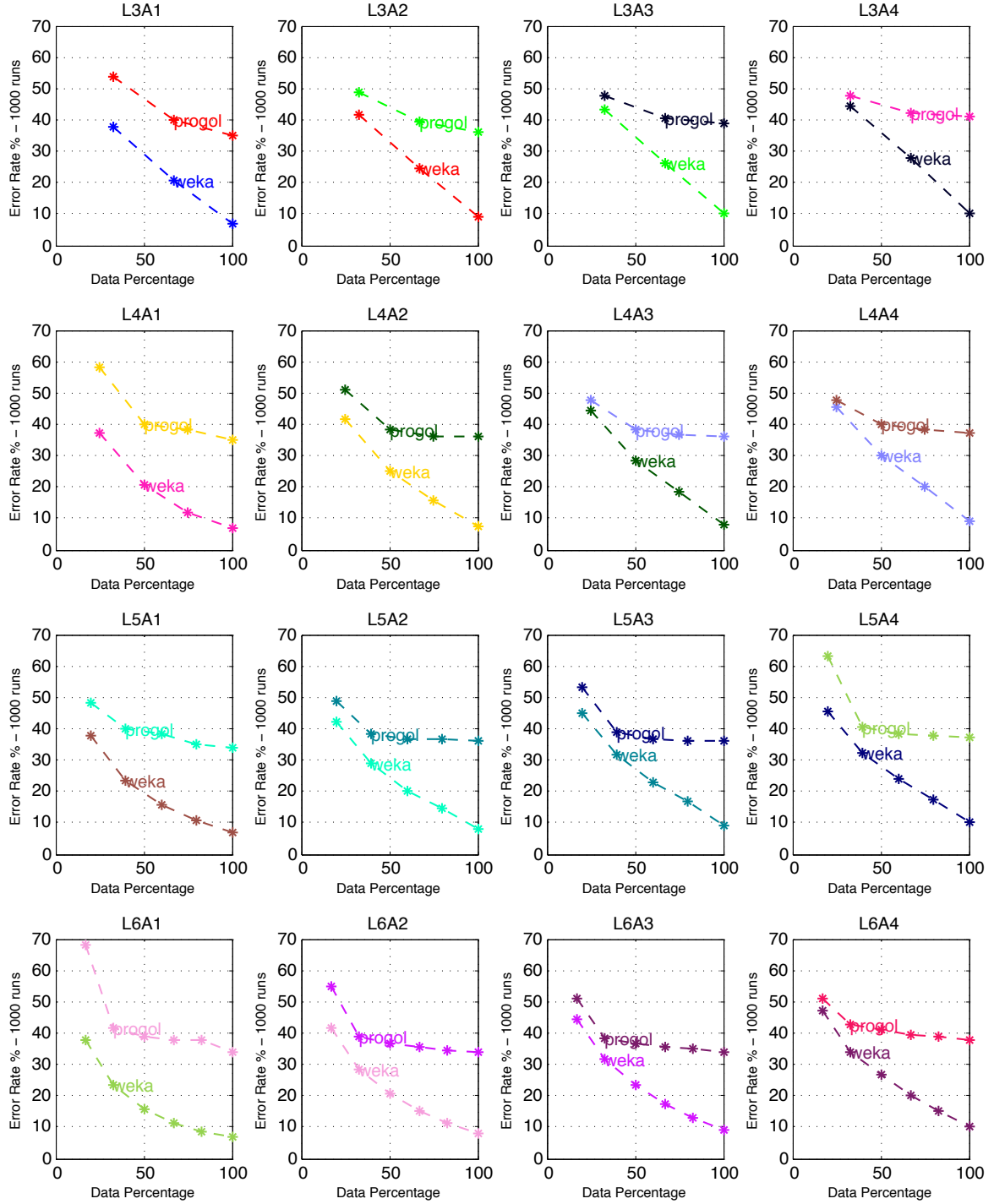


Figure 9.3: Temporal Error Rates in Weka and Progol

was able to extract more useful additional information comparing to Progol. This is due to Progol's Occam compression which also reflects on the fact that extra information contributes less to the decrease of error rates in Progol compared to Weka. As the graphs depict, error rates

in Progol solutions decrease for only a few percents when more than half of the information is revealed, in contrast to the drastic decrease of error rates for the same intervals in Weka.

9.2.6 A note on Occam's Razor

William of Ockham, regarded as one of most prominent logicians of the Middle Ages, is best known for his methodological principle named as *Occam's Razor*. The principle is important in a wide range of fields such as metaphysics, logic, natural philosophy, theory of knowledge, etc. The application of this principle in learning theory is the shorter description length principle. It states that shorter theories should have preference to the longer ones, given the example coverage being equal. Translating Occam's Razor into machine learning terms: if all hypotheses are equal in terms of their predictive accuracy, then it is rational to adopt the simplest one for scientific purposes [11, 119]. However, the principle is not set-in stone as a distinction is often made between various facets of simplicity, or at certain medical cases, a counterargument known as Hickam's dictum [59] refuses to wield the razor. Hickam's dictum principle refuses the diagnostic parsimony in medical domain by stating that patients can have as many diseases and the doctors should strive for testing all the hypotheses.

DIPs provide a concrete reason to reject the usage of Occam's razor. Remember that DIPs are dynamic. As such, at any time during an investigation, it is best to have a top suspect/diagnosis (or set of suspects/diagnoses) for the case at hand. Note that the extra constraint learned from the past cases might over-constrain the current case, i.e., it rules out all the suspects. To cater for this, GH calculates the proportion of predicates in the extra constraint that can be applied to each suspect. For example, suppose the learned constraint in a medical investigation is:

```
diagnosis(A):- pred1(_,_,A), pred5(_,A), pred8(_,_,_,_,A,_).
```

This is true of all the diagnoses in the case studies, but it might not be true of any suspect in the current case. However, suppose `pred1(_,_,A)` is true for suspect `s1`, but not `pred5(_,A)` or `pred8(_,_,_,_,A,_)`. In this case, as described earlier in §7.3, we assign a score of $1/3 = 0.33$ to suspect `s1`. Suppose further that suspect `s2` has two of the three predicates true about

him/her. Hence suspect **s2** should score 0.66 for the extra constraint, and be seen as more likely to be the cause of illness (guilty) than suspect **s1**.

It is clear that in this sense, we can rank all the suspects in terms of the parts of the constraints for the current case, and the parts of the extra constraint that they satisfy. This ranking will give us a set of top candidates to investigate further, and we can categorize suspects together if they have the same score with respect to the extra constraint. It is also clear that, if there are n suspects and m predicates in the extra constraint, and the literals are randomly true for the suspects, then we should expect $m + 1$ classes containing $\frac{n}{m+1}$ suspects. $m + 1$ classes are comprised of suspects satisfying $\{0, 1, \dots, m\}$ predicates.

Imagine, in a problem with 10 suspects, there are a number of extra constraints which are output by the machine learning system, all of which perfectly classify guilty and non-guilty suspects. Suppose the most general of these (MG) contains two predicates, and the most specific (MS) contains five predicates. In the most likely cause of illness (guilty) category for MG (i.e., the suspects with the highest scores for satisfying predicates from the extra constraint), there will be 10/3 suspects (probably 3). However, using the MS constraint, there will be 10/6 (probably 2) suspects. At this point, the doctor has to narrow down the diagnoses to just one, in order to suggest a treatment. Hence, in this case, the most specific case will be the best in that it narrows down the possibilities the quickest. In addition, if the doctor finds out that the most likely candidates are not the cause of illness, his later options will facilitate the diagnostic process in that the suspects are still in better hierarchy compared to the ones categorized by the most general rule; suspects not satisfying the diagnosis either hold 0, 20, 40, 60 or 90% or likelihood, whereas the most general rule with 2 predicates will categorize those suspects in either 0 or 50% likelihood degrees. Therefore, it would be best here to discard Occams advice and choose the most complicated solution.

In summary, taking the most complex of the learned rules will be the best choice when the doctor/detective wants to narrow down his/her options. Therefore, one can argue that the notion of one machine learning system being “better” than another should be in terms of how general/specific the learned ruleset is, depending on the context at hand; noting that a system

should be deemed better than another by default if it gets better predictive accuracy.

Finding the most specific rules will be problematic for software such as Progol, which chooses the simplest (in terms of Kolmogorov Complexity [129]) of the learned rulesets with the highest predictive accuracy [84]. It is similarly a problem for HR2, given that it usually constructs the simplest concepts before the more complicated ones (though it does not discard anything, but it will need more time to come up with the more complicated solutions). This drawback of HR2 has been considered and removed in the latest iteration of HR series, i.e., HR3 [36].

9.2.7 Rule Evaluation Measures in Association Rule Mining

Recalling from §2.7.1, association rule mining is primarily used for highlighting relationships between variables and to extract useful and interesting patterns in transactional databases. Transactional databases are comprised of a set of items and a transaction identifier. Apriori is considered to be the most widely used algorithm since its first introduction in [6]. The algorithm considers a framework which evaluates and extracts the rules based on two interestingness measures: support and confidence. We used Apriori to extract the rules in DIP case studies due to its popularity and availability as a built-in algorithm in Weka. The Apriori and in particular the interestingness factors for evaluating and ranking the associated rules have been the subject of a number of surveys [70, 9, 72].

The main interestingness criteria for evaluating a rule is (a) generality and (b) reliability [53, 125]. Support or coverage represents the generality and confidence can depict the reliability of a rule. However, there are other widely used criteria such as lift [19], conviction [18] and leverage [101] that we have used in order to compare with “confidence” in mining DIP rules. In all these methods, minimum support is initially used to weed out the uninteresting rules and other factors are then employed to extract the highly correlated ones. In this section we present the comparison between these factors in various DIP types.

We evaluated the aforementioned interestingness factors in terms of their execution time and the error rate of their extracted rules. Throughout the experiments, minimum support was set

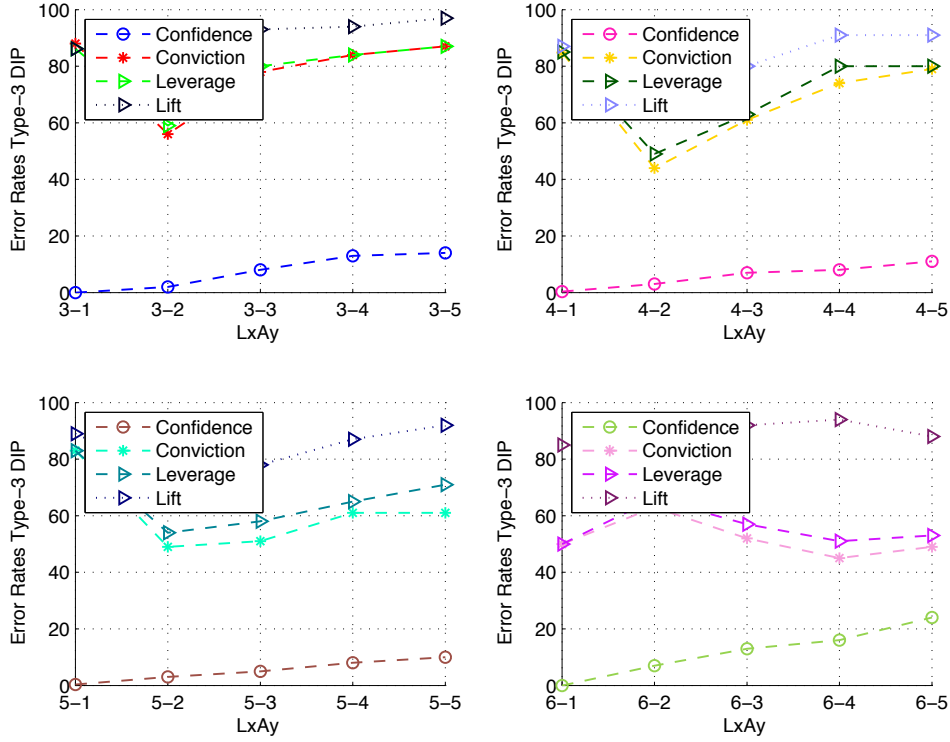


Figure 9.4: Percentage error rates of various interestingness measures, averaged over 1000 Type-3 investigation problems requiring solutions of the form LxAy

to 0.1 and minimum thresholds for confidence, conviction, leverage and lift were set to 0.99, 1.1, 0.01 and 1.1 respectively. Experimental results for type-1 and type-2 DIPs showed that all evaluating factors perform the same in terms of the error rate. Besides, all the factors showed negligible differences in terms of execution time. However, performance differences of various interestingness factors are highlighted in type-3 DIPs. Their rules introduced errors to the solutions and the rates also varied for different problem sets. The first reason why interestingness measures perform better in type-1 and type-2 DIPs compared to type-3 is that type-3 problem sets contain noise; case studies are generated while considering the reality factors.

According to figure 9.4, *confidence* is the best rule evaluation for type-3 DIPs and *lift* is the worst evaluation parameter. Lift of a rule $X \rightarrow Y$: $\frac{P(X,Y)}{P(X).P(Y)}$ highlights to which extend X and Y are *not* independent. Lift is known to be susceptible to noise when the dataset is small, similar to case studies of a DIP. It allocates high values to the rules with low probability that appear a few times by chance [19]. However, this cannot be always the right evaluation factor of the embedded rules in our DIPs as the rules may or may not be satisfied by a large number

of suspects. In summary, lift plays as a biased factor towards less frequent rules.

As we see in figure 9.4, leverage factor performs slightly better than lift. It measures the difference between the antecedent and the consequent of the rule (when both of them appear) and their expectations if they were independent, i.e., Leverage of a rule $X \rightarrow Y$: $P(X, Y) - (P(X).P(Y))$. In other words, it expresses how many more times a rule appears compared to the individual occurrences of antecedent and consequent. Therefore, leverage imposes a frequency constraint which may cause this factor to suffer from the rare frequency of itemsets which is the case for DIPs.

Conviction is known to be an alternative to confidence in that it can capture the direction of associations in a better way. However, as seen in figure 9.4, it still performs worse than confidence in mining DIP rules. Conviction of a rule $X \rightarrow Y$: $\frac{P(X).P(\neg Y)}{P(X, \neg Y)}$, highlights the frequency by which a rule fails to predict correctly if X and Y were independent divided by the observed number of incorrect predictions. Confidence of a rule $X \rightarrow Y$: $P(Y|X)$ simply calculates the probability of a consequent under the condition that the antecedent appears. According to the results, this criteria is more suitable to be employed by GH to mine the association rules.

The aforementioned measures are in the domain of *objective* measures of interestingness that are based on probability, statistics and information theory. We note that there are other measures that, unlike objective factors, incorporate the users' background knowledge and consider the goals of the data mining task. These include *subjective* and *semantic-based* measures [53]. Extensive work has been conducted to survey, evaluate and compare the interestingness measures of generated rules with no widespread acceptance of the formal definition of interestingness. The most striking in the comparison between interestingness factors is the lack of implication upon which one factors is always preferred to another. Researchers generally agree that none of the interestingness measures can be considered superior to the others and choosing each, solely depends of the individual task [53, 70, 125].

9.3 Commentary on the Results

The results demonstrate the success of GH in solving DIP problems with various types of difficulties. They also depict the weakness of the ILP system, Progol, in terms of execution time and error rate. We already discussed in §9.2.6 why Progol cannot extract the most specific rules due to employing Occam compression. However, Progol is inefficient in finding the solutions even in type-2 DIPs where only a single rule is embedded in the problems. The only problem set in which Progol performed better than the other systems was when the solution was in the form of LxA1. There are a number of possible reasons that explain the performance of Progol and association rule mining (Apriori) which we attempt to discuss in this section.

Recalling from chapter 2 (§2.8), *relational* representations are frameworks that can express a variable number of entities and their constituent relationships, and *logical* representations are when the relational frameworks are grounded in or derived from first order logic [42]. Logical and relational learning thus falls in the category of learning and mining within expressive knowledge representation which encompasses relational and first order logic. It, particularly, targets problems where multiple entities and their relationships should be learned, similar to what GH aims to learn from the case studies of dynamic investigation problems. ILP as discussed in chapter 2 (§2.8.1), is a suitable approach for learning such problems.

Progol [84, 90], as a landmark implementation of ILP, was chosen to be employed by GH in that Progol has so far inspired, and its algorithm is still at the core of many ILP systems. Recalling from chapter 2 (§2.8.1), Progol attempts to learn a set of hypothesized clauses H from background knowledge B and a set of examples E with the use of mode declarations which restrict the hypothesis space. Progol aims to find clauses in H in such a way that each $h \in H$ along with the background B entails at least a positive examples $e \in E$, i.e., $B, h \models e$. The next step is to make H exclude all the negative examples (Progol is normally more flexible in order to compensate for noisy data). With the use of contraposition, the above entailment can be rearranged to $B, \neg e \models \neg h$ which allows the hypotheses to be derived from B and e with the use of standard Prolog theorem proving methods. Omitting the already described details of Progol for the sake of brevity, Progol attempts to find a subset of solutions for H from the

clauses that θ -subsume the most specific clause (bottom-clause).

Progol, employing a top-down approach, initially selects an example e and constructs the most specific clause, $\perp_e$, that covers e , i.e., $B, \neg e \models \neg \perp_e \models \neg h$. Progol then applies a refinement operator which considers hypothesis h that θ -subsumes $\perp_e$, hence the partially ordered set (lattice) is as follows: $\perp_e \preceq h \prec \square$. We defined the symbol $\preceq$, subsumption and more general/specific hypothesis in §2.8.4. Progol uses mode declarations to restrict the search for clauses that subsume the bottom clause. The main drawback is that the $\perp_e$ and subsequently the search lattice may get too large, hence, only a fraction of the lattice can be searched [124]. Top-down algorithms such as Progol can limit the maximum complexity of learned clauses due to the search bias they undertake which favors simplicity. However, they tend to spend overwhelming amount of search to find the long clauses [120]. The results shown in this chapter also confirm the poor performance of Progol where the target clauses (solutions in the form of $LxAy$) are long and complex. There are systems such as ProGolem that try to tackle this problem using a bottom-up approach by which long clauses can be constructed easier [92]. The cardinality of Progol's search space, according to [84], is of the order shown in formula 9.1.

$$O(r|M|^{2ij(c+1)}) \quad (9.1)$$

Recalling from §2.8.1, M is the cardinality of mode declarations (for DIPs this number is equal to the number of predicates (facts) appearing in the problem description, as we assume that all predicates may appear in the body of the target clause), c is the user-defined maximum number of body literals, i is the number of iterations in the construction of the bottom clause, j is the number which bounds the occurrences of $+type$ and $-type$ in the mode declarations (arity) and r is the maximum depth of resolutions (unifications). It is clear from the formula that the complexity increases exponentially with the increase of body literals in the target concept. This primarily explains why Progol does not perform well in complex DIPs.

As discussed in §2.8, there are various representation formalisms that can be employed in learning problems. Attribute-value representations, multi-instance representations, relational representations and logical programs are formalisms that each are suitable and applied for par-

ticular learning problems in that each has a different representational power. Expressiveness of some formalisms such as logic programs comes at a computational cost. This explains why learning techniques built for those representations such as Prolog can be less efficient than techniques developed for learning in less expressive formalisms such as association rule mining in attribute-value representations. Less expressive learning techniques can be obtained by *downgrading* more expressive methods; the opposite method is known as *upgrading* [42]. One of the advantages of reducing the representations is that more traditional learning algorithms can be used, however, some data may get lost in transformation, and some hypotheses may be missed as solutions.

The algorithm we proposed in §4.5.3 is an example of downgrading. It flattens DIPs in Prolog notation to be suitable for association rule mining. The results shown in this chapter confirm that learning from DIP case studies can be more efficient using the less expressive representation (attribute-value). Association rule mining algorithms can find the hypotheses faster than Prolog because Prolog search space is much larger. Hence, it is more expensive to be searched as each hypothesis needs to be checked against all examples. Given d items, the search space for association rule mining in its most primitive method, “frequent item-set mining”, is calculated with order given in formula 9.2.

$$O(NMw) \text{ where } M = 2^d \quad (9.2)$$

N is the maximum number of transactions, w is the maximum number of items appearing in the transactions and 2^d is the number of possible candidate itemsets from d items. Translating this to DIPs, N will be the total number of facts appearing in the problem, M is the list of candidates generated from suspects and w is the maximum number of suspects appearing in a fact. Using the Apriori algorithm, employed by GH and described in §2.7.1, pruning techniques such as the anti-monotone property of support will reduce M . In addition, the Apriori algorithm can reduce the number of comparisons (NM) by using a hash tree data structure for storing candidate itemsets [6, 7]. Therefore, association rule mining is expected to be more efficient and less costly than using Prolog.

However, association rules are less expressive compared to hypothesis (propositional logic vs first order logic). Indeed, we needed to carefully flatten out our knowledge base using algorithm 4.4 to get a complete reduction to attribute-value form. This *propositionalization* [42] is not always possible/practical as inconsistencies and data loss may occur frequently so that many of the essential features of data may not be captured during the transformation. The results shown in this chapter are testament to the accuracy of algorithm 4.4 and its efficiency in representing investigation problems whereby the correct solutions were found in all type-1 and type-2 DIPs.

Even though the results illustrated the usefulness of association rule mining in learning the solutions from DIP case studies, one may argue the applicability of this algorithm in all relational learning problems. Researchers have devoted a lot of work on discovering association rules since the seminal paper by Agrawal et al. [6] and realized that association rule mining might not be suitable for all complex relational learning problems, particularly ones that cannot be easily represented by feature vectors and item sets. Algorithms in [47, 48] offer the flexibility required for finding relational association rules by unifying an ILP approach and association rule discovery.

9.4 Conclusions

In this chapter we illustrated and discussed the results of the experiments mentioned in chapter 8, table 8.1. We showed the results of association rule mining (Apriori), ILP (Progol) and Automated Theory Formation systems (HR3) in learning DIP embedded rules. The efficiency of bespoke systems was mainly measured by calculating the execution time and the error rates of the solutions that each rule set produces after being applied to the current case of a DIP. Their performance was evaluated in various experiments that were designed to gauge the GH's ability in solving DIPs with various levels of difficulty.

We also explained the Occam's Razor principle and discussed why systems such as Progol consider this principle in evaluating the merit of a learned rule. We explained why type-3 DIPs can be used as a yardstick to assess Progol's performance in comparison with other employed

systems. We also discussed why it is best to discard Occam's advice and choose the most specific hypothesis while solving DIPs.

In this chapter, the downgrading technique was discussed and we explained why such a method was used to transform DIP case studies in Prolog notation to attribute-value representations. By downgrading the expressiveness of DIPs, we were able to use association rule mining to learn from the case studies which is considered to be a relational problem. Following the association rule mining analysis, we discussed various interestingness measure employed by the Apriori algorithm. It was shown why the confidence factor is more appropriate for solving DIPs comparing to conviction, lift and leverage.

Finally, Progol's search space was compared to that for association rule mining. We explained why Progol struggles in finding lengthy complex rules. We introduced Progol+ which is simply a program that systematically runs Progol with various parameters and mode declarations. It enabled us to find the optimum parameters to fine-tune Progol with respect to DIPs. We showed how Progol+ improved Progol's performance in certain DIP types.

Chapter 10

Conclusions

Throughout this work we were inspired by the stories of legendary Sherlock Holmes [4] and also medical drama television series, House, M.D. [3]. In House stories, for instance, a patient with a mysterious malady is normally presented to the protagonist Dr. House and his team. When all other doctors have lost hope on the patient, genius Dr. House with his astute logical reasoning takes the case and strives to solve it. During the diagnostics process, he tries to form a plausible theory which best explains patients symptoms. He then tries to prove the theory by performing different medical tests. The interesting part is when throughout the diagnostics he discerns contradictory results. This makes him eliminate some of the symptoms as irrelevant or add unseen symptoms to the theory that he believes are implicitly embodied in the problem and may be manifested later on. Different reasoning approaches can be identified in his diagnostics process.

Drawing upon those stories, in this thesis, we introduced a type of hybrid AI problems which models to some extent generic situations which may arise in, say, medical diagnosis or the solving of a crime. We used the term *investigation problems* to denote such problem where the goal is to chose a guilty party, responsible for a phenomenon occurring in a medical or criminal investigation, from a number of suspects. That is, there are a number of possible diagnoses/suspects (*candidates*), and the problem is to use the facts of the case to rank them in order of increasing likelihood of being the cause of the illness/guilty of the crime (which we

call the *target candidate*).

Such ranking often leads to further medical tests/police enquiries focusing on the most likely candidates, which will bring to light further information about the current case. Hence, we use the term *dynamic investigation problem* (DIP) to describe a series of such problems to be solved. Solving each problem entails using the facts of the case, coupled with prior knowledge about the domain to narrow down the candidates to just one. Hence, a natural way to model such problems is as a constraint satisfaction problem (CSP), with one variable which takes one of n values, each representing a candidate, and the facts of the case acting as the constraints. Solving the case means finding a value to assign to the variable which doesn't break the constraints.

Often, however, not all the essential information is readily available, hence these problems are best modeled as partial CSPs. As such, especially during the early stages of the investigation, there will be no outright solution, and the constraints in the CSP need to be used to rank the candidates for further investigation. Additional relevant information can often be found in related past cases, from which regularities can be observed and utilised, and consultation of previous case studies is part of the investigation process. Hence dynamic investigation problems are hybrid machine-learning/constraint-solving problems, and as such are more realistic and of interest to the wider AI community.

10.1 Summary of the Thesis

Over the course of this thesis we largely focused on the (a) formal definition and exploration, (b) generation and (c) solution of *dynamic Investigation Problems*. We built a framework which performs *investigative reasoning*, that is a framework in which a combination of reasoning techniques are incorporated in order to tackle dynamic investigation problems. The reasoning systems involved are HR, Progol, Progol+, Weka's association rule mining, SICStus CLPFD, and on occasions *Pe-Pl*. All the bespoke systems are harnessed to form the constituents of an ad-hoc system we developed and named as GH. It has been named after the fictional medical investigator Gregory House, although his namesake of Sherlock Holmes would equally suffice.

The GH system, in summary, is responsible for generating and solving DIPs. It also benefits from a series of translators and partial solvers we designed to ensure the appropriate interaction between GH various incorporated reasoning systems. Here, we review the objectives and contributions we highlighted in chapter 1 (§1.1).

One of our main challenges was to mathematically define dynamic investigation problems thereby we could artificially generate them for our investigations. Chapter 3 provides a formal definition of investigations problems (IP) and dynamic investigation problems (DIP). In addition, we have illustrated and discussed a case study of a DIP which paved our way for automatic generation of such problems in later chapters.

In chapter 4 we have introduced algorithms for generating DIPs with various levels of difficulty. We have also elucidated translation algorithms by which we transformed our problems into the syntaxes of ad-hoc systems incorporated in our reasoning framework. Following the generation process, another challenge was to impose more dissimilarity to the current case in that DIPs had to be more realistic. Reality factors have been introduced in chapter 5 by which we enabled GH to generate more realistic scenarios.

We have tried to show the solution methodologies employed in tackling the preliminary case studies of Aunt Agatha and Cluedo problems in chapter 6. Solving those problem were among the first attempts at investigating ill-formed problems that neither of the standalone AI techniques such as constraint solving, machine learning or automated theorem proving could solve. In chapter 7 we have formalized what we mean by a solution to a DIP and discussed the ways in which GH proceeds in the absence of a single or a set of definite answers. We have also discussed how GH utilizes a combination of AI systems such as HR, Weka Association Rule Mining, Progol, Progol+, SICStus CLPFD and *Pe-pl* [40], in order to tackle DIPs

In chapter 8 we have discussed how to evaluate GH's proposed solution in order to demonstrate the usefulness of GH as a combined reasoning system for solving DIPs. We have mentioned the benefits of generating thousands of problems and explained the methodologies for calculating the error rate of the solutions. In addition, we have demonstrated the details of DIP types in terms of their levels of difficulty. We also elaborated on the methodology by which GH's

temporal solutions are evaluated and proposed to the user. In general, chapter 8 provides a detailed description of the experiments we designed to gauge the GH's ability in solving DIPs.

Chapter 9 provides a comprehensive comparison between various systems incorporated by GH. We have shown and compared the results of association rule mining (Apriori), ILP (Progol, Progol+) and Automated Theory Formation systems (HR2 and HR3) in learning DIPs embedded rules. In addition, we justified the efficiency of the bespoke systems in different scenarios.

10.2 Future directions

GH employs various and disparate reasoning systems to solve DIPs, hence, reasoning mechanisms involved in GH play a crucial part in GH's processes. As such, improvements to GH are mostly predicated on employing more efficient and/or more applicable reasoning systems. There are a few approaches that can enhance the reasoning mechanism of GH, among which the probabilistic reasoning can be one of the most promising ones. This direction has already been discussed in chapter 2 (§2.9) in that we performed few experiments with *Pe-Pl* with its results shown in §7.4. We already explained throughout this thesis that similar past cases bear relevant information to the current case from which regularities can often be observed and utilized in the current case. We demonstrated this by learning the profile of the guilty party from the case studies, however, such rules are evaluated with respect to the past cases. “Probabilistic inductive logic programming” can allow GH to rank the likelihood of the learned rules with respect to the current case.

10.2.1 Argumentation

In chapters 3 and 5 we discussed the case studies of a DIP and tried to make the problems more difficult by modifying the past cases generation algorithm. As there is no a-priori need for the case studies to be consistent with each other, in future work, we can look at “argumentation” to undertake conflict resolution and improve the intelligence of the system. Argumentation is

a powerful mechanism that has attracted interest in tackling the challenges occur in situations where the information is incomplete or even inconsistent. Argumentation can be employed by GH to evaluate the possible conclusions derived from a set of arguments and counter-arguments. Having originally a wide application in the realms of philosophy and law, argumentations has recently been used for decision making in the field of medicine [39, 77] which is one of the domains of our investigative reasoning.

There are a few other directions that we have been considering for our future work. Due to some preliminary analysis we performed on those approaches with respect to DIPs, we discuss them in more details as follows. We believe they may significantly improve the GH reasoning mechanism.

10.2.2 Theory Revision

Recalling from chapter 2, in ILP, we induce new knowledge based on learning examples. Given a set of information as background knowledge and a set of positive and negative examples, an ILP system will consequently generate a hypothesis. This way of knowledge acquisition will form a general theory based on the given information. Often, however, the existing knowledge is incorrect, that is, it contains pieces of information that contradict each other known as inconsistencies. In addition, in many cases, not all the essential information is readily available. In such scenarios, the knowledge is considered as incomplete or obscure. The process of automatically improving an existing knowledge based on the machine learning methods is called theory revision or theory refinement [109]. The difference between an ILP and a theory revision system is that the former one is primarily focused on generating a theory and does not address the issue of modifying the incorrect knowledge, while the latter is already provided with a theory along with background knowledge and a set of examples and it mainly concerns with improving the theory.

FORTE (First Order Revision of Theories from Examples) [109] is a knowledge refinement system we aim to use in our future work. We once employed FORTE for solving the Aunt Agatha problem described in §3.3 as a machine learning system. We aimed to investigate if

FORTE can infer the hypothesis *killer hates the victim and is never richer than the victim*. The input to the system contained the positive and negative examples along with the axioms describing the problem. It was needed to tweak our problem to be in the format acceptable by FORTE's different functionality blocks. Some of the blocks are as follows: *Theory translator* is to translate between a native representation of a theory and the representation required by FORTE, which is first order function-free Prolog [49]. Adjusting the Agatha problem to FORTE, the system managed to add: `killer(A,B):- hates(A, B), \+ richer(A, B).`

Implementing FORTE for a simple problem like this, where we only needed to generate a hypothesis based on the positive and negative examples was a redundant hardwork. One could realize that FORTE has more detailed representation formalities compared to Progol. For instance, segmenting the information for fundamental domain theory block, or considering language bias block. Achieving the same result as Progol required more effort from the user. However, there is a specification in FORTE which makes it different to Progol. As we described in the beginning of this section, the advantage of any theory revision system to an ILP system is the ability to revise and correct knowledge. For our DIP solver, we can focus on Progol and HR for axiom formation and/or reformation, but theory revision can be used to find the inconsistencies in DIP's background knowledge.

10.2.3 A cognitive approach for solving DIPs

Cognitive science is an approach for studying mind and intelligence and involves various subjects such as artificial intelligence, psychology, neuroscience, linguistics and philosophy. It initially started systematically right after experimental psychology emerged. Researchers tried to develop theories of mind, based on “complex representations” and “computational procedures”. However, the outlook of cognitive science gradually became diverse based on the contributions of the experts from different fields to form a unifying theoretical idea of how mind and intelligence work. Cognitive science as an interdisciplinary approach tries to study the way humans perform deductive reasoning, forming and applying concepts, mental imaging, analogical problem solving, and so on and so forth [126]. In the future we are interested in unifying GH's incorporated

reasoning system with a cognitively inspired model of axiom formation and reformulation.

In order to solve a problem, according to Gestalt psychologists [78], we need to search for a relation between aspects of a problem situation in order to form a structural understanding. This helps us to comprehend the way problem aspects fit together for satisfying a certain goal. In general, this may involve reorganizing the problem elements, creating new solutions for the problem (producing) or applying past solutions to similar problems (reproducing). More precisely, problem solving can be the outcome of thinking as hypothesis testing (inductive reasoning), logically drawn conclusions (deductive reasoning) and being influenced by experiences based on analogs and models. Having studied inductive and deductive reasoning and the ways we employ such approaches, we will expand our work by analyzing possibilities of solving IPs and DIPs using *analogy*.

An *analog* problem is the one which is similar to another problem in structure, but not necessarily in the story line. The story line is the overall definition of the problem and its characters. Two stories can have different story lines. The characters in both stories can be different while having the same structure. Let us consider the Fortress and Dunckers radiation problems described fully in [63]. In this problem, many roads are radiated outward from the fortress like spokes of a wheel. The General needs to capture the fortress given the fact that if all the soldiers attack the fortress from the same road, the mines mounted on the road will detonate and hence the soldiers will all get killed. On the contrary, the mines could tolerate a small group of soldiers without being exploded. Hence, the general decides to split up the army into small groups and sends each group to the beginning of a different road. This resolved the problem and the General captured the fortress. Another example fitting our discussion here, is a doctor that aims to solve a problem similar to the fortress but different on the surface known as radiation problem. This problem depicts a condition in which a patient needs to be operated by a special ray thereby destroying his malignant tumor. If the ray reaches the tumor with high intensity, the tumor will be destroyed, however, unfortunately, the healthy tissues, exposed to tumor, will also be destroyed. We can call the radiation problem as *target* and the fortress as *analog*. Knowing about the solution of the analog, the doctor gets an insight to divide the ray into segments with lower density and then converging them on the tumor. Structural similarities

between the two problems fall into *problem description* and *solution* segments. In the former one, similarities between fortress and tumor, mined road and surrounding tissues, streams of attacking soldiers and rays with different intensities are highlighted. In the solution part, the resemblance is identified between the small groups of soldiers converge at fortress and less intensity rays focused on the tumor.

In order to solve a problem using analogies, we need to consider three major aspects: (a) the recognition process in which a solver needs to memorize and recall the analog, (b) the abstraction process that comes after selecting a supposedly similar problem where the solver should pick the underlying similar goals and the common solutions and (c) the mapping that involves analogical reasoning to find an appropriate connection between the target and the analog problems [63, 78].

Having briefly talked about problem solving using analogy, we aim to investigate a method by which a theory of embodied mathematics is found. In this method, the authors, Lakoff and Núñez, propose that cognitive mechanisms and everyday concepts can direct our unconscious conceptualizations of technical ideas which results in producing mathematics [69]. The relation between this work and our problem solving is the analogy Lakoff and Núñez use between the characterization of ordinary nonmathematical ideas and mathematical ideas. In summary, they believe that nonmathematical thought can create mathematical understanding. This analogical approach which is based on conceptual metaphors will hopefully give us an insight for solving the problems this project addresses. In order to understand this approach, it is essential to grasp the “metaphor” concept upon which Lakoff and Núñez build their theory.

Abstraction, as we mentioned in aspects of analogical problem solving, is to help us understand the similar goals and common solutions of two problems which are in different domains. Metaphors, apart from being a figure of speech, “are basic means by which abstract thought is possible” [69]. Lakoff and Núñez use metaphorical cross-domain mappings in order to perform abstract reasoning. Imagine a metaphor “Categories Are Containers”. Through this metaphor one can understand the similarities between a bounded region and a category, or an object inside a bounded region and a category member. Applying this mapping from an analog problem

(source domain) to a target problem (target domain), we can observe inference patterns such as Modes Ponens. In a source domain (container schema inference), given two container schemas, if A is in B and B is in C , then A is in C . This can be mapped to target domain as two given categories A and B and an entity X , If A is in B and X is in A , then X is in B .

There are two conceptual metaphors particularly for characterizing mathematical ideas: grounding and linking metaphors. Grounding metaphors yield basic, directly grounding ideas such as addition as adding objects to a collection, subtraction as taking objects from a collection, groups and sets as containers, and also members of sets as objects in a container. Grounding ideas reveals the structural relationship across domains. Linking metaphors on the other hand are useful for producing “sophisticated ideas” such as numbers as points on the line and geometric figures as algebraic equations. Linking metaphors conceptualizes ideas in the other domains of mathematics in terms of arithmetic, and can probably become a pivotal point of our work by using the Lakoff and Núñez theory in our research. Imagine we have a medical diagnosis problem that is structurally similar to a previously solved criminology problem. In a series of steps involving problem identification, we need to find the structural similarities between these problems (ground metaphors) and map the similar abstract from the solved problem to the target domain (linking metaphors). Therefore, it is essential to find out how we can produce metaphors that enable the analogy to take place. For the DIP problems in particular, the analogies can hold between: (i) past cases and the current problem, (ii) problem at every stage with its previous stage(s). Therefore, first of all, we need to determine and identify analogs and subsequently use Lakoff and Núñez linking metaphors for mapping the analogs, thereby learning the profile of the guilty party from one domain and utilize it in another domain.

GH has shown promising results in solving dynamic investigation problems and we will continue to develop GH in order to understand better its full potential. We believe that GH is a useful tool for practitioners who which to use investigative reasoning and we hope to find, eventually, more researchers embracing the idea of using a combined reasoning framework for dynamic investigation problems. These problems can eventually model more sophisticated medical or criminal scenarios and GH can hopefully be used to help solving such problems in real life.

Appendix A

Generation Appendix

A.1 Example of Cluedo Past Cases

```
% case 1
angry(case-1,green). angry(case-1,mustard).angry(case-1,plum).angry(case-1,white).is_short
(case-1,green).is_short(case-1,mustard).is_short(case-1,white).is_shortest(case-1,white).is_tall
(case-1,peacock).is_tall(case-1,plum).is_tall(case-1,scarlett).is_tallest(case-1,scarlett).missing_
weapon(case-1,dagger).missing_weapon(case-1,rope).murder_scene(case-1,hall).murder_weapon(case
-1,rope).murderer(case-1,plum).potential_murder_weapon(case-1,rope).suspect_room(case-1,hall).
suspect_room(case-1,kitchen).loves(case-1,green,scarlett).loves(case-1,mustard,scarlett).loves
(case-1,plum,scarlett).was_found(case-1,candlestick,diningroom).was_found(case-1,leadpipe,patio).
was_found(case-1,revolver,spa).was_found(case-1,spanner,guesthouse).

% case 2
angry(case-2,green).angry(case-2,peacock).angry(case-2,scarlett).angry(case-2,white).is_short(case-2,p
eacock).is_short(case-2,scarlett).is_short(case-2,white).is_shortest(case-2,white).is_tall(case-2,gree
n).is_tall(case-2,mustard).is_tall(case-2,plum).is_tallest(case-2,plum).missing_weapon(case-2,candles
tick).missing_weapon(case-2,rope).murder_scene(case-2,diningroom).murder_weapon(case-2,rope).murderer(
case-2,green).potential_murder_weapon(case-2,rope).suspect_room(case-2,diningroom).suspect_room(case-2
,guesthouse).loves(case-2,green,scarlett).loves(case-2,peacock,scarlett).loves(case-2,white,scarlett).
was_found(case-2,dagger,livingroom).was_found(case-2,leadpipe,hall).was_found(case-2,revolver,spa).was
_found(case-2,spanner,kitchen).

% case 3
angry(case-3,green).angry(case-3,plum).angry(case-3,scarlett).angry(case-3,white).is_short(case-3,gree
n).is_short(case-3,scarlett).is_short(case-3,white).is_shortest(case-3,white).is_tall(case-3,mustard).
is_tall(case-3,peacock).is_tall(case-3,plum).is_tallest(case-3,plum).missing_weapon(case-3,candlesti
ck).missing_weapon(case-3,leadpipe).murder_scene(case-3,patio).murder_weapon(case-3,candlestick).murde
rer(case-3,plum).potential_murder_weapon(case-3,candlestick).suspect_room(case-3,livingroom).
```

```

suspect_room(case-3,patio).loves(case-3,green,scarlett).loves(case-3,plum,scarlett).loves(case-3,white,scarlett
).was_found(case-3,dagger,observatory).was_found(case-3,revolver,kitchen).was_found(case-3,rope,guesth
ouse).was_found(case-3,spanner,diningroom).
% case 4
angry(case-4,green).angry(case-4,mustard).angry(case-4,plum).angry(case-4,scarlett).is_short(case-4,
green).is_short(case-4,plum).is_short(case-4,white).is_shortest(case-4,white).is_tall(case-4,mustard).
is_tall(case-4,peacock).is_tall(case-4,scarlett).is_tallest(case-4,scarlett).missing_weapon(case-4,
candlestick).missing_weapon(case-4,spanner).murder_scene(case-4,diningroom).murder_weapon(case-4,
candlestick).murderer(case-4,mustard).potential_murder_weapon(case-4,candlestick).suspect_room(case-4
,diningroom).suspect_room(case-4,hall).loves(case-4,green,scarlett).loves(case-4,mustard,scarlett).lov
es(case-4,plum,scarlett).was_found(case-4,dagger,observatory).was_found(case-4,leadpipe,spa).was_found
(case-4,revolver,theater).was_found(case-4,rope,kitchen).
% case 5
angry(case-5,peacock).angry(case-5,plum).angry(case-5,scarlett).angry(case-5,white).is_short(case-5,pe
acock).is_short(case-5,plum).is_short(case-5,scarlett).is_shortest(case-5,plum).is_tall(case-5,green).
is_tall(case-5,mustard).is_tall(case-5,white).is_tallest(case-5,green).missing_weapon(case-5,revolver
).missing_weapon(case-5,rope).murder_scene(case-5,theater).murder_weapon(case-5,revolver).murderer(cas
e-5,white).potential_murder_weapon(case-5,revolver).suspect_room(case-5,diningroom).suspect_room(case-
5,theater).loves(case-5,peacock,scarlett).loves(case-5,plum,scarlett).loves(case-5,white,scarlett).was
_found(case-5,candlestick,kitchen).was_found(case-5,dagger,spa).was_found(case-5,leadpipe,livingroom).
was_found(case-5,spanner,patio).

```

A.2 Generic DIP: A set of past cases

```

% case 1
%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_).
answer(s18).pred1(s10,s11,s17,s13,s19).pred1(s13,s16,s12,s19,s17).pred2(s17).pred2(s10).
pred2(s12).pred2(s15).pred2(s16).pred3(s10,s19,s14,s11,s11).pred3(s11,s17,s11,s19,s10).
pred3(s13,s13,s12,s15,s17).pred3(s11,s12,s11,s16,s14).pred3(s16,s12,s15,s15,s12).pred3
(s15,s19,s18,s15,s19).pred4(s19).pred4(s17).pred4(s18).pred5(s14).pred5(s15).pred6(s10,
s12,s17,s11,s14).pred6(s11,s13,s14,s15,s19).pred6(s13,s15,s17,s12,s10).pred7(s17,s15,
s14,s10,s16).pred7(s12,s10,s11,s17,s15).pred7(s14,s11,s19,s12,s16).pred8(s16,s14,s19,s10).
pred8(s11,s15,s13,s16).pred9(s14,s19).pred9(s16,s10).pred10(s12,s13,s14,s10,s12).
pred10(s16,s11,s13,s18,s15).pred10(s13,s15,s13,s13,s11).pred10(s14,s10,s16,s16,s11).
pred10(s16,s17,s16,s19,s12).pred10(s10,s13,s12,s18,s14).
% case 2
%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_).
answer(s24).pred2(s21).pred2(s25).pred2(s29).pred3(s28,s23,s24,s29,s27).
pred3(s26,s23,s26,s20,s28).pred3(s25,s25,s28,s28,s29).pred3(s27,s29,s25,s21,s22).
pred3(s29,s29,s21,s25,s25).pred3(s29,s23,s25,s25,s27).pred4(s29).pred4(s24).
pred4(s22).pred5(s26).pred6(s21,s25,s20,s29,s26).pred8(s21,s28,s20,s29).
pred9(s25,s20).pred9(s23,s26).pred10(s29,s27,s23,s22,s20).pred10(s23,s29,s21,s24,s26).
pred10(s20,s25,s26,s20,s27).pred10(s22,s26,s26,s27,s28).pred10(s22,s29,s21,s24,s25).
pred10(s23,s29,s29,s23,s28).

```

```

% case 3
%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_,_X,_).
answer(s37).pred2(s36).pred2(s32).pred3(s30,s31,s37,s33,s38).pred3(s33,s32,s34,s32,s34).
pred3(s39,s39,s35,s33,s38).pred3(s39,s30,s33,s35,s35).pred3(s36,s34,s36,s32,s32).
pred3(s38,s32,s35,s30,s39).pred4(s32).pred4(s37).pred4(s30).pred5(s31).pred5(s33).
pred5(s35).pred8(s32,s33,s31,s35).pred9(s32,s36).pred9(s30,s35).pred10(s38,s32,s34,s37,s35).
pred10(s33,s33,s34,s39,s34).pred10(s30,s32,s30,s31,s33).pred10(s38,s30,s38,s38,s30).
pred10(s34,s39,s35,s32,s35).pred10(s35,s35,s31,s37,s35).

% case 4
%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_,_X,_).
answer(s41).pred1(s43,s44,s45,s42,s47).pred2(s45).pred2(s47).pred3(s49,s40,s45,s49,s49).
pred3(s45,s45,s41,s49,s44).pred3(s48,s45,s44,s43,s45).pred3(s49,s43,s49,s47,s49).
pred3(s42,s40,s44,s47,s48).pred3(s44,s40,s43,s47,s47).pred4(s40).pred4(s46).pred4(s41).
pred5(s42).pred6(s42,s44,s45,s49,s46).pred7(s44,s46,s49,s48,s47).pred7(s45,s40,s46,s42,s48).
pred7(s46,s47,s45,s49,s44).pred8(s42,s45,s46,s43).pred10(s44,s49,s44,s42,s42).
pred10(s46,s47,s45,s47,s48).pred10(s43,s46,s43,s48,s43).pred10(s49,s43,s49,s41,s42).
pred10(s49,s46,s48,s41,s44).pred10(s40,s43,s49,s46,s42).

% case 5
%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_,_X,_).
answer(s50).pred2(s54).pred3(s54,s59,s55,s55,s57).pred3(s55,s53,s59,s59,s59).
pred3(s51,s55,s50,s56,s56).pred3(s53,s57,s57,s52,s57).pred3(s58,s53,s55,s59,s51).
pred3(s52,s56,s53,s51,s59).pred4(s50).pred4(s58).pred4(s56).pred5(s54).pred5(s55).
pred6(s53,s51,s55,s54,s52).pred6(s58,s52,s57,s53,s51).pred6(s59,s52,s55,s56,s53).
pred6(s57,s53,s51,s54,s58).pred7(s51,s54,s58,s56,s59).pred7(s58,s52,s51,s53,s54).
pred7(s52,s59,s53,s58,s56).pred7(s52,s54,s58,s53,s57).pred8(s59,s52,s57,s55).
pred8(s58,s54,s59,s56).pred8(s58,s52,s54,s56).pred9(s52,s59).pred9(s51,s55).
pred10(s55,s55,s54,s50,s58).pred10(s51,s55,s59,s54,s53).pred10(s54,s58,s52,s51,s52).
pred10(s56,s59,s52,s50,s52).pred10(s54,s55,s57,s52,s57).pred10(s57,s54,s59,s58,s56).

```

A.3 Translation Suite: DIP past cases in Progol format

```

% Translation of past cases for Progol
:- set(h,1000)?
:- set(i,1000)?
:- set(nodes, 1000)?
:- set(c,3)?

% mode declarations
:- modeb(*,murderer(+suspect))?
:- modeb(*, pred1(-suspect,-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred2(-suspect))?
:- modeb(*, pred3(-suspect,-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred4(-suspect))?
:- modeb(*, pred5(-suspect))?

```

```

:- modeb(*, pred6(-suspect,-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred7(-suspect,-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred8(-suspect,-suspect,-suspect,-suspect))?
:- modeb(*, pred9(-suspect,-suspect))?
:- modeb(*, pred10(-suspect,-suspect,-suspect,-suspect,-suspect))?

suspect(s10).suspect(s11).suspect(s12).suspect(s13).suspect(s14).suspect(s15).suspect(s16).
suspect(s17).suspect(s18).suspect(s19).suspect(s20).suspect(s21).suspect(s22).suspect(s23).
suspect(s24).suspect(s25).suspect(s26).suspect(s27).suspect(s28).suspect(s29).suspect(s30).
suspect(s31).suspect(s32).suspect(s33).suspect(s34).suspect(s35).suspect(s36).suspect(s37).
suspect(s38).suspect(s39).suspect(s40).suspect(s41).suspect(s42).suspect(s43).suspect(s44).
suspect(s45).suspect(s46).suspect(s47).suspect(s48).suspect(s49).suspect(s50).suspect(s51).
suspect(s52).suspect(s53).suspect(s54).suspect(s55).suspect(s56).suspect(s57).suspect(s58).
suspect(s59).

%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_,_).
pred1(s10,s11,s17,s13,s19).pred1(s13,s16,s12,s19,s17).pred2(s17).pred2(s10).pred2(s12).
pred2(s15).pred2(s16).pred3(s10,s19,s14,s11,s11).pred3(s11,s17,s11,s19,s10).
pred3(s13,s13,s12,s15,s17).pred3(s11,s12,s11,s16,s14).pred3(s16,s12,s15,s15,s12).
pred3(s15,s19,s18,s15,s19).pred4(s19).pred4(s17).pred4(s18).pred5(s14).pred5(s15).
pred6(s10,s12,s17,s11,s14).pred6(s11,s13,s14,s15,s19).pred6(s13,s15,s17,s12,s10).
pred7(s17,s15,s14,s10,s16).pred7(s12,s10,s11,s17,s15).pred7(s14,s11,s19,s12,s16).
pred8(s16,s14,s19,s10).pred8(s11,s15,s13,s16).pred9(s14,s19).pred9(s16,s10).pred10(s12,
s13,s14,s10,s12).pred10(s16,s11,s13,s18,s15).pred10(s13,s15,s13,s13,s11).pred10(s14,s10
,s16,s16,s11).pred10(s16,s17,s16,s19,s12).pred10(s10,s13,s12,s18,s14).

%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_,_).
pred2(s21).pred2(s25).pred2(s29).pred3(s28,s23,s24,s29,s27).pred3(s26,s23,s26,s20,s28).
pred3(s25,s25,s28,s28,s29).pred3(s27,s29,s25,s21,s22).pred3(s29,s29,s21,s25,s25).
pred3(s29,s23,s25,s25,s27).pred4(s29).pred4(s24).pred4(s22).pred5(s26).pred6(s21,s25,
s20,s29,s26).pred8(s21,s28,s20,s29).pred9(s25,s20).pred9(s23,s26).pred10(s29,s27,s23,
s22,s20).pred10(s23,s29,s21,s24,s26).pred10(s20,s25,s26,s20,s27).pred10(s22,s26,s26,
s27,s28).pred10(s22,s29,s21,s24,s25).pred10(s23,s29,s29,s23,s28).

%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_,_).
pred2(s36).pred2(s32).pred3(s30,s31,s37,s33,s38).pred3(s33,s32,s34,s32,s34).pred3(s39,s39,s35,s33,s38).
pred3(s39,s30,s33,s35,s35).pred3(s36,s34,s36,s32,s32).pred3(s38,s32,s35,s30,s39).pred4(s32).
pred4(s37).pred4(s30).pred5(s31).pred5(s33).pred5(s35).pred8(s32,s33,s31,s35).pred9(s32,s36).
pred9(s30,s35).pred10(s38,s32,s34,s37,s35).pred10(s33,s33,s34,s39,s34).pred10(s30,s32,s30,s31,s33).
pred10(s38,s30,s38,s38,s30).pred10(s34,s39,s35,s32,s35).pred10(s35,s35,s31,s37,s35).

%Constraints: answer(X):- pred3(_,_X,_,_), pred4(X), pred10(_,_X,_,_).
pred1(s43,s44,s45,s42,s47).pred2(s45).pred2(s47).pred3(s49,s40,s45,s49,s49).pred3(s45,s45,s41,s49,s44).
pred3(s48,s45,s44,s43,s45).pred3(s49,s43,s49,s47,s49).pred3(s42,s40,s44,s47,s48).pred3(s44,s40,s43,s47,
s47).pred4(s40).pred4(s46).pred4(s41).pred5(s42).pred6(s42,s44,s45,s49,s46).pred7(s44,s46,s49,s48,s47).
pred7(s45,s40,s46,s42,s48).pred7(s46,s47,s45,s49,s44).pred8(s42,s45,s46,s43).pred10(s44,s49,s44,s42,s42).
pred10(s46,s47,s45,s47,s48).pred10(s43,s46,s43,s48,s43).pred10(s49,s43,s49,s41,s42).pred10(s49,s46,s48,
s41,s44).pred10(s40,s43,s49,s46,s42).

```

```
%Constraints: answer(X):- pred3(_,_ ,X,_ ,_) , pred4(X) , pred10(_,_ ,_,X,_ ) .
pred2(s54) .pred3(s54,s59,s55,s55,s57) .pred3(s55,s53,s59,s59,s59) .pred3(s51,s55,s50,s56,s56) .
pred3(s53,s57,s57,s52,s57) .pred3(s58,s53,s55,s59,s51) .pred3(s52,s56,s53,s51,s59) .pred4(s50) .
pred4(s58) .pred4(s56) .pred5(s54) .pred5(s55) .pred6(s53,s51,s55,s54,s52) .pred6(s58,s52,s57,s53,s51) .
pred6(s59,s52,s55,s56,s53) .pred6(s57,s53,s51,s54,s58) .pred7(s51,s54,s58,s56,s59) .pred7(s58,s52,s51,
s53,s54) .pred7(s52,s59,s53,s58,s56) .pred7(s52,s54,s58,s53,s57) .pred8(s59,s52,s57,s55) .pred8(s58,s54,
s59,s56) .pred8(s58,s52,s54,s56) .pred9(s52,s59) .pred9(s51,s55) .pred10(s55,s55,s54,s50,s58) .pred10(s51,
s55,s59,s54,s53) .pred10(s54,s58,s52,s51,s52) .pred10(s56,s59,s52,s50,s52) .pred10(s54,s55,s57,s52,s57) .
pred10(s57,s54,s59,s58,s56) .

% Positive Examples
answer(s18) .answer(s24) .answer(s37) .answer(s41) .answer(s50) .
```

A.4 Translation Suite: DIP past cases in HR format

```
random1
suspect(A)
ascii:@A@ is suspect
prolog:suspect(@A@)
suspect(s10) .suspect(s11) .suspect(s12) .suspect(s13) .suspect(s14) .suspect(s15) .suspect(s16) .
suspect(s17) .suspect(s18) .suspect(s19) .suspect(s20) .suspect(s21) .suspect(s22) .suspect(s23) .
suspect(s24) .suspect(s25) .suspect(s26) .suspect(s27) .suspect(s28) .suspect(s29) .suspect(s30) .
suspect(s31) .suspect(s32) .suspect(s33) .suspect(s34) .suspect(s35) .suspect(s36) .suspect(s37) .
suspect(s38) .suspect(s39) .suspect(s40) .suspect(s41) .suspect(s42) .suspect(s43) .suspect(s44) .
suspect(s45) .suspect(s46) .suspect(s47) .suspect(s48) .suspect(s49) .suspect(s50) .suspect(s51) .
suspect(s52) .suspect(s53) .suspect(s54) .suspect(s55) .suspect(s56) .suspect(s57) .suspect(s58) .
suspect(s59) .

random2
answer(A)
ascii:@A@ is answer
prolog:murderer(@A@)
answer(A) -> suspect(A)
answer(s18) .answer(s24) .answer(s37) .answer(s41) .answer(s50) .

random3
pred1(A,B,D,E,F)
ascii:pred1 @A@,@B@,@D@,@E@,@F@
prolog:pred1(@A@,@B@,@D@,@E@,@F@)
pred1(A,B,D,E,F) -> suspect(A)
pred1(A,B,D,E,F) -> suspect(B)
pred1(A,B,D,E,F) -> suspect(D)
pred1(A,B,D,E,F) -> suspect(E)
pred1(A,B,D,E,F) -> suspect(F)
pred1(s10,s11,s17,s13,s19) .pred1(s13,s16,s12,s19,s17) .pred1(s43,s44,s45,s42,s47) .
```

```

random4
pred2(A)
ascii:pred2 @A@
prolog:pred2(@A@)
pred2(A) -> suspect(A)
pred2(s17).pred2(s10).pred2(s12).pred2(s15).pred2(s16).pred2(s21).pred2(s25).
pred2(s29).pred2(s36).pred2(s32).pred2(s45).pred2(s47).pred2(s54).

random5
pred3(A,B,D,E,F)
ascii:pred3 @A@,@B@,@D@,@E@,@F@
prolog:pred3(@A@,@B@,@D@,@E@,@F@)
pred3(A,B,D,E,F) -> suspect(A)
pred3(A,B,D,E,F) -> suspect(B)
pred3(A,B,D,E,F) -> suspect(D)
pred3(A,B,D,E,F) -> suspect(E)
pred3(A,B,D,E,F) -> suspect(F)
pred3(s10,s19,s14,s11,s11).pred3(s11,s17,s11,s19,s10).pred3(s13,s13,s12,s15,s17).
pred3(s11,s12,s11,s16,s14).pred3(s16,s12,s15,s15,s12).pred3(s15,s19,s18,s15,s19).
pred3(s28,s23,s24,s29,s27).pred3(s26,s23,s26,s20,s28).pred3(s25,s25,s28,s28,s29).
pred3(s27,s29,s25,s21,s22).pred3(s29,s29,s21,s25,s25).pred3(s29,s23,s25,s25,s27).
pred3(s30,s31,s37,s33,s38).pred3(s33,s32,s34,s32,s34).pred3(s39,s39,s35,s33,s38).
pred3(s39,s30,s33,s35,s35).pred3(s36,s34,s36,s32,s32).pred3(s38,s32,s35,s30,s39).
pred3(s49,s40,s45,s49,s49).pred3(s45,s45,s41,s49,s44).pred3(s48,s45,s44,s43,s45).
pred3(s49,s43,s49,s47,s49).pred3(s42,s40,s44,s47,s48).pred3(s44,s40,s43,s47,s47).
pred3(s54,s59,s55,s55,s57).pred3(s55,s53,s59,s59,s59).pred3(s51,s55,s50,s56,s56).
pred3(s53,s57,s57,s52,s57).pred3(s58,s53,s55,s59,s51).pred3(s52,s56,s53,s51,s59).

random6
pred4(A)
ascii:pred4 @A@
prolog:pred4(@A@)
pred4(A) -> suspect(A)
pred4(s19).pred4(s17).pred4(s18).pred4(s29).pred4(s24).pred4(s22).pred4(s32).
pred4(s37).pred4(s30).pred4(s40).pred4(s46).pred4(s41).pred4(s50).pred4(s58).
pred4(s56).

random7
pred5(A)
ascii:pred5 @A@
prolog:pred5(@A@)
pred5(A) -> suspect(A)
pred5(s14).pred5(s15).pred5(s26).pred5(s31).pred5(s33).pred5(s35).pred5(s42).
pred5(s54).pred5(s55).

random8
pred6(A,B,D,E,F)
ascii:pred6 @A@,@B@,@D@,@E@,@F@
prolog:pred6(@A@,@B@,@D@,@E@,@F@)
pred6(A,B,D,E,F) -> suspect(A)
pred6(A,B,D,E,F) -> suspect(B)

```

```

pred6(A,B,D,E,F) -> suspect(D)
pred6(A,B,D,E,F) -> suspect(E)
pred6(A,B,D,E,F) -> suspect(F)
pred6(s10,s12,s17,s11,s14).pred6(s11,s13,s14,s15,s19).pred6(s13,s15,s17,s12,s10).
pred6(s21,s25,s20,s29,s26).pred6(s42,s44,s45,s49,s46).pred6(s53,s51,s55,s54,s52).
pred6(s58,s52,s57,s53,s51).pred6(s59,s52,s55,s56,s53).
pred6(s57,s53,s51,s54,s58).

random9
pred7(A,B,D,E,F)
ascii:pred7 @A@,@B@,@D@,@E@,@F@
prolog:pred7(@A@,@B@,@D@,@E@,@F@)
pred7(A,B,D,E,F) -> suspect(A)
pred7(A,B,D,E,F) -> suspect(B)
pred7(A,B,D,E,F) -> suspect(D)
pred7(A,B,D,E,F) -> suspect(E)
pred7(A,B,D,E,F) -> suspect(F)
pred7(s17,s15,s14,s10,s16).pred7(s12,s10,s11,s17,s15).pred7(s14,s11,s19,s12,s16).
pred7(s44,s46,s49,s48,s47).pred7(s45,s40,s46,s42,s48).pred7(s46,s47,s45,s49,s44).
pred7(s51,s54,s58,s56,s59).pred7(s58,s52,s51,s53,s54).pred7(s52,s59,s53,s58,s56).
pred7(s52,s54,s58,s53,s57).

random10
pred8(A,B,D,E)
ascii:pred8 @A@,@B@,@D@,@E@
prolog:pred8(@A@,@B@,@D@,@E@)
pred8(A,B,D,E) -> suspect(A)
pred8(A,B,D,E) -> suspect(B)
pred8(A,B,D,E) -> suspect(D)
pred8(A,B,D,E) -> suspect(E)
pred8(s16,s14,s19,s10).pred8(s11,s15,s13,s16).pred8(s21,s28,s20,s29).pred8(s32,s33,s31,s35).
pred8(s42,s45,s46,s43).pred8(s59,s52,s57,s55).pred8(s58,s54,s59,s56).pred8(s58,s52,s54,s56).

random11
pred9(A,B)
ascii:pred9 @A@,@B@
prolog:pred9(@A@,@B@)
pred9(A,B) -> suspect(A)
pred9(A,B) -> suspect(B)
pred9(s14,s19).pred9(s16,s10).pred9(s25,s20).pred9(s23,s26).pred9(s32,s36).pred9(s30,s35).
pred9(s52,s59).pred9(s51,s55).

random12
pred10(A,B,D,E,F)
ascii:pred10 @A@,@B@,@D@,@E@,@F@
prolog:pred10(@A@,@B@,@D@,@E@,@F@)

```

```

pred10(A,B,D,E,F) -> suspect(A)
pred10(A,B,D,E,F) -> suspect(B)
pred10(A,B,D,E,F) -> suspect(D)
pred10(A,B,D,E,F) -> suspect(E)
pred10(A,B,D,E,F) -> suspect(F)
pred10(s12,s13,s14,s10,s12).pred10(s16,s11,s13,s18,s15).pred10(s13,s15,s13,s13,s11).
pred10(s14,s10,s16,s16,s11).pred10(s16,s17,s16,s19,s12).pred10(s10,s13,s12,s18,s14).
pred10(s29,s27,s23,s22,s20).pred10(s23,s29,s21,s24,s26).pred10(s20,s25,s26,s20,s27).
pred10(s22,s26,s26,s27,s28).pred10(s22,s29,s21,s24,s25).pred10(s23,s29,s29,s23,s28).
pred10(s38,s32,s34,s37,s35).pred10(s33,s33,s34,s39,s34).pred10(s30,s32,s30,s31,s33).
pred10(s38,s30,s38,s38,s30).pred10(s34,s39,s35,s32,s35).pred10(s35,s35,s31,s37,s35).
pred10(s44,s49,s44,s42,s42).pred10(s46,s47,s45,s47,s48).pred10(s43,s46,s43,s48,s43).
pred10(s49,s43,s49,s41,s42).pred10(s49,s46,s48,s41,s44).pred10(s40,s43,s49,s46,s42).
pred10(s55,s55,s54,s50,s58).pred10(s51,s55,s59,s54,s53).pred10(s54,s58,s52,s51,s52).
pred10(s56,s59,s52,s50,s52).pred10(s54,s55,s57,s52,s57).pred10(s57,s54,s59,s58,s56).

```

A.5 Translation Suite: DIP past cases in Weka format

```

@relation murderer
@attribute pred1 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5,
15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345
@attribute pred2 1
@attribute pred3 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5,
15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345
@attribute pred4 1
@attribute pred5 1
@attribute pred6 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5,
15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345
@attribute pred7 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234,
5,
15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345
@attribute pred8 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234, 1234
@attribute pred9 1, 2, 12
@attribute pred10 1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234,
1234, 5,
15, 25, 125, 35, 135, 235, 1235, 45, 145, 245, 1245, 345, 1345, 2345, 12345
@attribute murderer yes, no
@data
?,?,3,1,?,?,?,4,yes
1,1,15,?,?,15,24,4,2,124,no
2,?,1345,?,?,14,23,1,?,25,no
35,1,25,1,?,3,14,?,?,2,no
14,?,12,?,?,12,?,3,?,1234,no
45,?,245,1,?,5,3,3,2,4,no
2,1,14,?,?,?,5,14,1,134,no
3,1,235,?,?,24,14,?,?,135,no
?,1,134,?,1,24,25,2,?,25,no
?,?,35,?,1,35,13,2,1,135,no
?,?,3,1,?,?,?,4,yes
?,1,34,?,?,1,?,1,?,3,no
?,1,12345,?,?,2,?,?,1,25,no
?,1,1245,1,?,4,?,4,?,123,no
?,?,1345,?,?,?,2,?,5,no
?,?,2,?,?,?,?,1,134,no

```

?,1,134,?,1,24,25,2,?,25,no
 ?,?,35,?,1,35,13,2,1,135,no
 ?,?,3,1,?,?,?,?,4,yes
 ?,1,34,?,?,1,?,1,?,3,no
 ?,1,12345,?,?,2,?,?,1,25,no
 ?,1,1245,1,?,4,?,4,?,123,no
 ?,?,1345,?,?,?,2,?,5,no
 ?,?,2,?,?,?,?,1,134,no
 ?,?,15,?,?,?,?,?,245,no
 ?,?,13,?,1,5,?,?,2,235,no
 ?,?,4,?,?,3,?,3,2,145,no
 ?,?,5,1,?,?,?,?,14,no
 ?,?,3,1,?,?,?,?,4,yes
 ?,1,13,?,?,?,?,2,?,no
 ?,1,245,1,?,?,?,1,1,24,no
 ?,?,124,1,?,?,?,?,1,1235,no
 ?,?,2,?,1,?,?,3,?,34,no
 ?,?,134,?,1,?,?,2,?,125,no
 ?,?,15,?,?,?,?,?,134,no
 ?,?,235,?,?,?,?,?,135,no
 ?,?,125,?,?,?,?,?,24,no
 ?,?,345,?,1,?,?,4,2,1235,no
 ?,?,3,1,?,?,?,?,4,yes
 1,?,234,?,?,?,?,4,?,1235,no
 2,?,135,?,?,2,15,?,?,135,no
 3,1,1235,?,?,3,13,2,?,3,no
 4,?,1,?,1,1,4,1,?,45,no
 5,1,45,?,?,?,25,?,?,24,no
 ?,?,1345,?,?,4,34,?,?,123,no
 ?,?,2,1,?,?,2,?,?,1,no
 ?,?,15,?,?,?,45,?,?,345,no
 ?,?,?,1,?,5,123,3,?,124,no
 ?,?,3,1,?,?,?,?,4,yes
 ?,1,1,?,1,4,25,23,?,1234,no
 ?,?,2345,?,?,1,25,13,2,23,no
 ?,?,1234,?,1,3,?,4,2,12,no
 ?,?,235,?,?,13,5,3,?,135,no
 ?,?,123,?,?,1245,34,?,?,5,no
 ?,?,145,?,?,235,13,?,1,14,no
 ?,?,245,1,?,4,45,4,?,15,no
 ?,?,14,?,?,25,12,2,1,345,no
 ?,?,1,1,?,15,134,1,?,245,no

Appendix B

Solution Appendix

B.1 Solving Aunt Agatha Problem

The process in which OTTER proves “Aunt Agatha killed herself” is shown in the following sections. The input file is shown in §B.1.1 and output in §B.1.2.

B.1.1 Aunt Agatha: OTTER input

```
set(auto).
assign(max_seconds, 10).
assign(max_mem, 1000000).
formula_list(usable).
lives_in_mansion(agatha).
lives_in_mansion(butler).
lives_in_mansion(charles).
all x (hates(agatha, x) -> -(hates(charles, x))).
hates(agatha, agatha).
hates(agatha, charles).
all x (hates(agatha, x) -> hates(butler, x)).
all x y (killed(x, y) -> hates(x, y)).
all x y (killed(x, y) -> -(richer(x, y))).
all x (-(richer(x, agatha)) -> hates(butler, x)).
all x (-(hates(agatha, x)) <-> x = butler).
exists x (lives_in_mansion(x) & killed(x, agatha)).
all x (lives_in_mansion(x) -> (x = agatha | x = butler | x = charles)).
% by commenting the following statement, Otter can not prove the final goal
% either of the following statements -- output from HR
all x y (hates(x, y) -> -(all z (hates(x, z)))).
% -(exists x (all y (hates(x, y)))).
% Negate the statement you need to prove
-killed(agatha, agatha).
%-(killed(butler, agatha) | killed(agatha,agatha)).
end_of_list.
```

B.1.2 Aunt Agatha: OTTER output

```

----- Otter 3.3f, August 2004 -----
The process was started by Ramin on Ramin-Ramezanis-iMac.local,
Sat Jun 22 17:53:56 2013
The command was "otter". The process ID is 1190.
set(auto).
  dependent: set(auto1).
  dependent: set(process_input).
  dependent: clear(print_kept).
  dependent: clear(print_new_demod).
  dependent: clear(print_back_demod).
  dependent: clear(print_back_sub).
  dependent: set(control_memory).
  dependent: assign(max_mem, 12000).
  dependent: assign(pick_given_ratio, 4).
  dependent: assign(stats_level, 1).
  dependent: assign(max_seconds, 10800).
assign(max_seconds,10).
assign(max_mem,1000000).
formula_list(usable).
lives_in_mansion(agatha).
lives_in_mansion(butler).
lives_in_mansion(charles).
all x (hates(agatha,x)-> -hates(charles,x)).
hates(agatha,agatha).
hates(agatha,charles).
all x (hates(agatha,x)->hates(butler,x)).
all x y (killed(x,y)->hates(x,y)).
all x y (killed(x,y)-> -richer(x,y)).
all x (-richer(x,agatha)->hates(butler,x)).
all x (-hates(agatha,x)<->x=butler).
exists x (lives_in_mansion(x)&killed(x,agatha)).
all x (lives_in_mansion(x)->x=agatha|x=butler|x=charles).
all x y (hates(x,y)-> -(all z hates(x,z))).
-killed(agatha,agatha).
end_of_list.
-----> usable clausifies to:
list(usable).
0 [] lives_in_mansion(agatha).
0 [] lives_in_mansion(butler).
0 [] lives_in_mansion(charles).
0 [] -hates(agatha,x)| -hates(charles,x).
0 [] hates(agatha,agatha).
0 [] hates(agatha,charles).
0 [] -hates(agatha,x)|hates(butler,x).
0 [] -killed(x,y)|hates(x,y).
0 [] -killed(x,y)| -richer(x,y).
0 [] richer(x,agatha)|hates(butler,x).
0 [] hates(agatha,x)|x=butler.
0 [] -hates(agatha,x)|x!=butler.
0 [] lives_in_mansion($c1).
0 [] killed($c1,agatha).
0 [] -lives_in_mansion(x)|x=agatha|x=butler|x=charles.
0 [] -hates(x,y)| -hates(x,$f1(x,y)).
0 [] -killed(agatha,agatha).
end_of_list.
SCAN INPUT: prop=0, horn=0, equality=1, symmetry=0, max_lits=4.
This is a non-Horn set with equality. The strategy will be
Knuth-Bendix, ordered hyper_res, factoring, and unit
deletion, with positive clauses in sos and nonpositive
clauses in usable.

```

```

dependent: set(knuth_bendix).
dependent: set(anl_eq).
dependent: set(para_from).
dependent: set(para_into).
dependent: clear(para_from_right).
dependent: clear(para_into_right).
dependent: set(para_from_vars).
dependent: set(eq_units_both_ways).
dependent: set(dynamic_demod_all).
dependent: set(dynamic_demod).
dependent: set(order_eq).
dependent: set(back_demod).
dependent: set(lrpo).
dependent: set(hyper_res).
dependent: set(unit_deletion).
dependent: set(factor).
-----> process usable:
** KEPT (pick-wt=6): 1 [] -hates(agatha,x)| -hates(charles,x).
** KEPT (pick-wt=6): 2 [] -hates(agatha,x)|hates(butler,x).
** KEPT (pick-wt=6): 3 [] -killed(x,y)|hates(x,y).
** KEPT (pick-wt=6): 4 [] -killed(x,y)| -richer(x,y).
** KEPT (pick-wt=6): 5 [] -hates(agatha,x)|x!=butler.
** KEPT (pick-wt=11): 6 [] -lives_in_mansion(x)|x=agatha|x=butler|x=charles.
** KEPT (pick-wt=8): 7 [] -hates(x,y)| -hates(x,$f1(x,y)).
** KEPT (pick-wt=3): 8 [] -killed(agatha,agatha).
-----> process sos:
** KEPT (pick-wt=2): 9 [] lives_in_mansion(agatha).
** KEPT (pick-wt=2): 10 [] lives_in_mansion(butler).
** KEPT (pick-wt=2): 11 [] lives_in_mansion(charles).
** KEPT (pick-wt=3): 12 [] hates(agatha,agatha).
** KEPT (pick-wt=3): 13 [] hates(agatha,charles).
** KEPT (pick-wt=6): 14 [] richer(x,agatha)|hates(butler,x).
** KEPT (pick-wt=6): 15 [] hates(agatha,x)|x=butler.
** KEPT (pick-wt=2): 16 [] lives_in_mansion($c1).
** KEPT (pick-wt=3): 17 [] killed($c1,agatha).
===== end of input processing =====
===== start of search =====
given clause #1: (wt=2) 9 [] lives_in_mansion(agatha).
given clause #2: (wt=2) 10 [] lives_in_mansion(butler).
given clause #3: (wt=2) 11 [] lives_in_mansion(charles).
given clause #4: (wt=2) 16 [] lives_in_mansion($c1).
given clause #5: (wt=3) 12 [] hates(agatha,agatha).
given clause #6: (wt=3) 13 [] hates(agatha,charles).
given clause #7: (wt=3) 17 [] killed($c1,agatha).
given clause #8: (wt=3) 22 [hyper,12,2] hates(butler,agatha).
given clause #9: (wt=3) 25 [hyper,13,2] hates(butler,charles).
given clause #10: (wt=3) 28 [hyper,17,3] hates($c1,agatha).
given clause #11: (wt=6) 14 [] richer(x,agatha)|hates(butler,x).
given clause #12: (wt=3) 41 [hyper,14,4,17] hates(butler,$c1).
given clause #13: (wt=6) 15 [] hates(agatha,x)|x=butler.
given clause #14: (wt=5) 50 [hyper,15,7,13] $f1(agatha,charles)=butler.
given clause #15: (wt=3) 60 [hyper,50,5,15,demod,51] butler=butler.
given clause #16: (wt=9) 18 [hyper,9,6,flip.2,flip.3]
agatha=agatha|butler=agatha|charles=agatha.
given clause #17: (wt=3) 64 [para_from,50.1.1,7.2.2,unit_del,13]
-hates(agatha,butler).
given clause #18: (wt=5) 52 [hyper,15,7,12] $f1(agatha,agatha)=butler.
given clause #19: (wt=6) 31 [para_into,17.1.1,6.2.1,unit_del,8,16,flip.1,flip.2]
butler=$c1|charles=$c1.
given clause #20: (wt=6) 54 [hyper,15,2] x=butler|hates(butler,x).
given clause #21: (wt=9) 20 [hyper,11,6] charles=agatha|charles=butler|charles=charles.
given clause #22: (wt=5) 80 [hyper,54,7,41] $f1(butler,$c1)=butler.
given clause #23: (wt=3) 93 [para_from,80.1.1,7.2.2,unit_del,41]
-hates(butler,butler).

```

```

given clause #24: (wt=3) 95 [para_into,93.1.2,31.1.1,unit_del,41] charles=$c1.
given clause #25: (wt=3) 106 [back_demod,67,demod,96,factor_simp] $c1=$c1.
given clause #26: (wt=8) 49 [hyper,15,7,15] $f1(agatha,x)=butler|x=butler.
given clause #27: (wt=3) 110 [back_demod,13,demod,96] hates(agatha,$c1).
given clause #28: (wt=5) 84 [hyper,54,7,22] $f1(butler,agatha)=butler.
given clause #29: (wt=5) 108 [back_demod,50,demod,96] $f1(agatha,$c1)=butler.
given clause #30: (wt=6) 66 [para_into,18.3.1,18.3.1,factor_simp,factor_simp,factor_simp]
agatha=agatha|butler=agatha.
given clause #31: (wt=11) 58 [para_from,15.2.1,7.2.2] -hates(x,y)|
-hates(x,butler)|hates(agatha,$f1(x,y)).
given clause #32: (wt=3) 115 [para_into,66.2.1,66.2.1,factor_simp,factor_simp]
agatha=agatha.
given clause #33: (wt=6) 112 [back_demod,1,demod,96] -hates(agatha,x)|
-hates($c1,x).
-----> EMPTY CLAUSE at 0.00 sec ----> 118 [hyper,112,12,28] $F.
Length of proof is 8. Level of proof is 5.
----- PROOF -----
1 [] -hates(agatha,x)| -hates(charles,x).
2 [] -hates(agatha,x)|hates(butler,x).
3 [] -killed(x,y)|hates(x,y).
4 [] -killed(x,y)| -richer(x,y).
6 [] -lives_in_mansion(x)|x=agatha|x=butler|x=charles.
7 [] -hates(x,y)| -hates(x,$f1(x,y)).
8 [] -killed(agatha,agatha).
12 [] hates(agatha,agatha).
14 [] richer(x,agatha)|hates(butler,x).
15 [] hates(agatha,x)|x=butler.
16 [] lives_in_mansion($c1).
17 [] killed($c1,agatha).
28 [hyper,17,3] hates($c1,agatha).
31 [para_into,17.1.1,6.2.1,unit_del,8,16,flip.1,flip.2]
butler=$c1|charles=$c1.
41 [hyper,14,4,17] hates(butler,$c1).
54 [hyper,15,2] x=butler|hates(butler,x).
80 [hyper,54,7,41] $f1(butler,$c1)=butler.
93 [para_from,80.1.1,7.2.2,unit_del,41] -hates(butler,butler).
96,95 [para_into,93.1.2,31.1.1,unit_del,41] charles=$c1.
112 [back_demod,1,demod,96] -hates(agatha,x)| -hates($c1,x).
118 [hyper,112,12,28] $F.
----- end of proof -----
Search stopped by max_proofs option.
===== end of search =====
----- statistics -----
clauses given          33
clauses generated      388
clauses kept           109
clauses forward subsumed 321
clauses back subsumed   44
Kbytes malloced        976
----- times (seconds) -----
user CPU time          0.00          (0 hr, 0 min, 0 sec)
system CPU time        0.00          (0 hr, 0 min, 0 sec)
wall-clock time        0             (0 hr, 0 min, 0 sec)
That finishes the proof of the theorem.
Process 1190 finished Sat Jun 22 17:53:56 2013

```

B.1.3 Aunt Agatha: CLPFD Translation

Below is the automatic translation of Aunt Agatha problem into CLPFD syntax, using the translation suite described in [23]

```
:-use_module(library(lists)).
:-use_module(library(clpfd)).
:-use_module(library(fdbg)).
:-use_module(solve_add).
:-initialization init.
init:-
  assert(solver_list_size_typed(30)),
  assert(types([[agatha,butler,charles]])),
  assert(pred_names_and_types([[hates,[1,1]],[killed,[1,1]],[lives,[1]],[richer,[1,1]]])).
solve(MT):-
  length(MT,30),
  domain(MT,0,1),
  % lives(agatha)
  p_1_1(MT,MT1),!,
  % lives(butler)
  p_2_1(MT1,MT2),!,
  % lives(charles)
  p_3_1(MT2,MT3),!,
  % all x ((~(hates(agatha,x))) | (~(hates(charles,x))))
  p_4_1(MT3,MT4),!,
  % hates(agatha,agatha)
  p_5_1(MT4,MT5),!,
  % hates(agatha,charles)
  p_6_1(MT5,MT6),!,
  % ~(hates(agatha,butler))
  p_7_1(MT6,MT7),!,
  % all x ((~(hates(agatha,x))) | (hates(butler,x)))
  p_8_1(MT7,MT8),!,
  % all x y ((~(killed(x,y))) | (hates(x,y)))
  p_9_1(MT8,MT9),!,
  % all x y ((~(killed(x,y))) | (~(richer(x,y))))
  p_10_1(MT9,MT10),!,
  % all x ((richer(x,agatha)) | (hates(butler,x)))
  p_11_1(MT10,MT11),!,
  % exists x ((lives(x)) & (killed(x,agatha)))
  p_12_1(MT11,MT12),!,
  write('GO!!'),nl,!,labeling([ff],MT12).
si:-solve(X),interpret_typed(X).
siw(X):-solve(X),write(X).
si(X):-solve(X),write(X),interpret_typed(X).
% lives(agatha)
p_1_1(MT,MT):-
  pred_var_at_typed(lives,[0],MT,PV1),
  PV1 #= 1.
% lives(butler)
p_2_1(MT,MT):-
  pred_var_at_typed(lives,[1],MT,PV1),
  PV1 #= 1.
% lives(charles)
p_3_1(MT,MT):-
  pred_var_at_typed(lives,[2],MT,PV1),
  PV1 #= 1.
% all x ((~(hates(agatha,x))) | (~(hates(charles,x))))
p_4_1(MT,MT1):-
  sub_sets_with_types([1],S),
  p_4_2(S,MT,MT1).
```

```

p_4_2([],MT,MT).
p_4_2([[V1]|Ss],MT,MT2):-
  p_4_3(V1,MT,MT1), % -(hates(agatha,x)) | -(hates(charles,x))
  p_4_2(Ss,MT1,MT2).
p_4_3(V1,MT,MT2):-
  p_4_4(V1,MT,MT1,RV1), % -(hates(agatha,x))
  p_4_5(V1,MT1,MT2,RV2), % -(hates(charles,x))
  RV1 # RV2.
p_4_4(V1,MT,MT,RV0):-
  pred_var_at_typed(hates,[0,V1],MT,PV1),
  PV1 #= 0 #<=> RV0.
p_4_5(V1,MT,MT,RV0):-
  pred_var_at_typed(hates,[2,V1],MT,PV1),
  PV1 #= 0 #<=> RV0.
  % hates(agatha,agatha)
p_5_1(MT,MT):-
  pred_var_at_typed(hates,[0,0],MT,PV1),
  PV1 #= 1.
  % hates(agatha,charles)
p_6_1(MT,MT):-
  pred_var_at_typed(hates,[0,2],MT,PV1),
  PV1 #= 1.
  % -(hates(agatha,butler))
p_7_1(MT,MT):-
  pred_var_at_typed(hates,[0,1],MT,PV1),
  PV1 #= 0.
  % all x ((-(hates(agatha,x))) | (hates(butler,x)))
p_8_1(MT,MT1):-
  sub_sets_with_types([1],S),
  p_8_2(S,MT,MT1).
p_8_2([],MT,MT).
p_8_2([[V2]|Ss],MT,MT2):-
  p_8_3(V2,MT,MT1), % -(hates(agatha,x)) | (hates(butler,x))
  p_8_2(Ss,MT1,MT2).
p_8_3(V2,MT,MT2):-
  p_8_4(V2,MT,MT1,RV1), % -(hates(agatha,x))
  p_8_5(V2,MT1,MT2,RV2), % hates(butler,x)
  RV1 # RV2.
p_8_4(V2,MT,MT,RV0):-
  pred_var_at_typed(hates,[0,V2],MT,PV1),
  PV1 #= 0 #<=> RV0.
p_8_5(V2,MT,MT,RV0):-
  pred_var_at_typed(hates,[1,V2],MT,PV1),
  PV1 #= 1 #<=> RV0.
  % all x y ((-(killed(x,y))) | (hates(x,y)))
p_9_1(MT,MT1):-
  sub_sets_with_types([1,1],S),
  p_9_2(S,MT,MT1).
p_9_2([],MT,MT).
p_9_2([[V3,V4]|Ss],MT,MT2):-
  p_9_3(V3,V4,MT,MT1), % -(killed(x,y)) | (hates(x,y))
  p_9_2(Ss,MT1,MT2).
p_9_3(V3,V4,MT,MT2):-
  p_9_4(V3,V4,MT,MT1,RV1), % -(killed(x,y))
  p_9_5(V3,V4,MT1,MT2,RV2), % hates(x,y)
  RV1 # RV2.
p_9_4(V3,V4,MT,MT,RV0):-
  pred_var_at_typed(killed,[V3,V4],MT,PV1),
  PV1 #= 0 #<=> RV0.
p_9_5(V3,V4,MT,MT,RV0):-
  pred_var_at_typed(hates,[V3,V4],MT,PV1),
  PV1 #= 1 #<=> RV0.
  % all x y ((-(killed(x,y))) | -(richer(x,y)))
p_10_1(MT,MT1):-
  sub_sets_with_types([1,1],S),
  p_10_2(S,MT,MT1).
p_10_2([],MT,MT).
p_10_2([[V5,V6]|Ss],MT,MT2):-
  p_10_3(V5,V6,MT,MT1), % -(killed(x,y)) | -(richer(x,y))
  p_10_2(Ss,MT1,MT2).

```

```

p_10_3(V5,V6,MT,MT2):-
  p_10_4(V5,V6,MT,MT1,RV1), % -(killed(x,y))
  p_10_5(V5,V6,MT1,MT2,RV2), % -(richer(x,y))
  RV1 # RV2.
p_10_4(V5,V6,MT,MT,RV0):-
  pred_var_at_typed(killed,[V5,V6],MT,PV1),
  PV1 #= 0 #<=> RV0.
p_10_5(V5,V6,MT,MT,RV0):-
  pred_var_at_typed(richer,[V5,V6],MT,PV1),
  PV1 #= 0 #<=> RV0.
  % all x ((richer(x,agatha)) | (hates(butler,x)))
p_11_1(MT,MT1):-
  sub_sets_with_types([1],S),
  p_11_2(S,MT,MT1).
p_11_2([],MT,MT).
p_11_2([V7|Ss],MT,MT2):-
  p_11_3(V7,MT,MT1), % (richer(x,agatha)) | (hates(butler,x))
  p_11_2(Ss,MT1,MT2).
p_11_3(V7,MT,MT2):-
  p_11_4(V7,MT,MT1,RV1), % richer(x,agatha)
  p_11_5(V7,MT1,MT2,RV2), % hates(butler,x)
  RV1 # RV2.
p_11_4(V7,MT,MT,RV0):-
  pred_var_at_typed(richer,[V7,0],MT,PV1),
  PV1 #= 1 #<=> RV0.
p_11_5(V7,MT,MT,RV0):-
  pred_var_at_typed(hates,[1,V7],MT,PV1),
  PV1 #= 1 #<=> RV0.
  % exists x ((lives(x)) & (killed(x,agatha)))
p_12_1(MT,MT2):-
  domain([EV8],0,2),
  p_12_2(EV8,MT,MT1), % (lives(x)) & (killed(x,agatha))
  append(MT1,[EV8],MT2).
p_12_2(EV8,MT,MT2):-
  p_12_3(EV8,MT,MT1), % lives(x)
  p_12_4(EV8,MT1,MT2). % killed(x,agatha)
p_12_3(EV8,MT,MT):-
  pred_var_at_typed(lives,[EV8],MT,PV1),
  PV1 #= 1.
p_12_4(EV8,MT,MT):-
  pred_var_at_typed(killed,[EV8,0],MT,PV1),
  PV1 #= 1.

```

B.2 Cluedo: CLPFD Translation and Solution

The complete translation and solution of the Cluedo scenario shown in chapter 6, figure 6.7, is as follow:

```
:-initialization init.
init:-
  assert(solver_list_size_typed(78)),
  assert(types([[green,mustard,peacock,plum,scarlett,white]])),
  assert(pred_names_and_types([[angry,[1]], [loves,[1,1]], [murderer,[1]], [person,[1]],
[short,[1]], [shortest,[1]], [tall,[1]], [tallest,[1]]])).
solve(MT):-
  length(MT,78),
  domain(MT,0,1),
  % person(mustard)
  p_1_1(MT,MT1),!,
  % person(peacock)
  p_2_1(MT1,MT2),!,
  % person(green)
  p_3_1(MT2,MT3),!,
  % person(white)
  p_4_1(MT3,MT4),!,
  % person(scarlett)
  p_5_1(MT4,MT5),!,
  % person(plum)
  p_6_1(MT5,MT6),!,
  % angry(mustard)
  p_7_1(MT6,MT7),!,
  % angry(scarlett)
  p_8_1(MT7,MT8),!,
  % angry(white)
  p_9_1(MT8,MT9),!,
  % angry(plum)
  p_10_1(MT9,MT10),!,
  % short(mustard)
  p_11_1(MT10,MT11),!,
  % short(green)
  p_12_1(MT11,MT12),!,
  % short(plum)
  p_13_1(MT12,MT13),!,
  % loves(white,scarlett)
  p_14_1(MT13,MT14),!,
  % loves(mustard,scarlett)
  p_15_1(MT14,MT15),!,
  % tall(peacock)
  p_16_1(MT15,MT16),!,
  % tall(white)
  p_17_1(MT16,MT17),!,
  % tall(scarlett)
  p_18_1(MT17,MT18),!,
  % tallest(scarlett)
  p_19_1(MT18,MT19),!,
  % shortest(green)
  p_20_1(MT19,MT20),!,
  % loves(plum,scarlett)
  p_21_1(MT20,MT21),!,
  % all x y ((-((tall(x)) & (loves(x,y)))) | (murderer(x)))
  p_22_1(MT21,MT22),!,
  write('GO!!!'),nl,!,labeling([ff],MT22).
si:-solve(X),interpret_typed(X).
siw(X):-solve(X),write(X).
si(X):-solve(X),write(X),interpret_typed(X).
```

```

% person(mustard)
p_1_1(MT,MT):-
  pred_var_at_typed(person,[1],MT,PV1),
  PV1 #= 1.
% person(peacock)
p_2_1(MT,MT):-
  pred_var_at_typed(person,[2],MT,PV1),
  PV1 #= 1.
% person(green)
p_3_1(MT,MT):-
  pred_var_at_typed(person,[0],MT,PV1),
  PV1 #= 1.
% person(white)
p_4_1(MT,MT):-
  pred_var_at_typed(person,[5],MT,PV1),
  PV1 #= 1.
% person(scarlett)
p_5_1(MT,MT):-
  pred_var_at_typed(person,[4],MT,PV1),
  PV1 #= 1.
% person(plum)
p_6_1(MT,MT):-
  pred_var_at_typed(person,[3],MT,PV1),
  PV1 #= 1.
% angry(mustard)
p_7_1(MT,MT):-
  pred_var_at_typed(angry,[1],MT,PV1),
  PV1 #= 1.
% angry(scarlett)
p_8_1(MT,MT):-
  pred_var_at_typed(angry,[4],MT,PV1),
  PV1 #= 1.
% angry(white)
p_9_1(MT,MT):-
  pred_var_at_typed(angry,[5],MT,PV1),
  PV1 #= 1.
% angry(plum)
p_10_1(MT,MT):-
  pred_var_at_typed(angry,[3],MT,PV1),
  PV1 #= 1.
% short(mustard)
p_11_1(MT,MT):-
  pred_var_at_typed(short,[1],MT,PV1),
  PV1 #= 1.
% short(green)
p_12_1(MT,MT):-
  pred_var_at_typed(short,[0],MT,PV1),
  PV1 #= 1.
% short(plum)
p_13_1(MT,MT):-
  pred_var_at_typed(short,[3],MT,PV1),
  PV1 #= 1.
% loves(white,scarlett)
p_14_1(MT,MT):-
  pred_var_at_typed(loves,[5,4],MT,PV1),
  PV1 #= 1.
% loves(mustard,scarlett)
p_15_1(MT,MT):-
  pred_var_at_typed(loves,[1,4],MT,PV1),
  PV1 #= 1.
% tall(peacock)
p_16_1(MT,MT):-
  pred_var_at_typed(tall,[2],MT,PV1),
  PV1 #= 1.
% tall(white)
p_17_1(MT,MT):-
  pred_var_at_typed(tall,[5],MT,PV1),
  PV1 #= 1.

```

```

% tall(scarlett)
p_18_1(MT,MT):-
  pred_var_at_typed(tall,[4],MT,PV1),
  PV1 #= 1.
% tallest(scarlett)
p_19_1(MT,MT):-
  pred_var_at_typed(tallest,[4],MT,PV1),
  PV1 #= 1.
% shortest(green)
p_20_1(MT,MT):-
  pred_var_at_typed(shortest,[0],MT,PV1),
  PV1 #= 1.
% loves(plum,scarlett)
p_21_1(MT,MT):-
  pred_var_at_typed(loves,[3,4],MT,PV1),
  PV1 #= 1.
% all x y ((-(tall(x)) & (loves(x,y)))) | (murderer(x))
p_22_1(MT,MT1):-
  sub_sets_with_types([1,1],S),
  p_22_2(S,MT,MT1).
p_22_2([],MT,MT).
p_22_2([V1,V2|Ss],MT,MT2):-
  p_22_3(V1,V2,MT,MT1), % -(tall(x)) & (loves(x,y)) | (murderer(x))
  p_22_2(Ss,MT1,MT2).
p_22_3(V1,V2,MT,MT2):-
  p_22_4(V1,V2,MT,MT1,RV1), % -(tall(x)) & (loves(x,y))
  p_22_5(V1,MT1,MT2,RV2), % murderer(x)
  RV1 # RV2.
p_22_4(V1,V2,MT,MT1,RV0):-
  p_22_6(V1,V2,MT,MT1,RV1), % (tall(x)) & (loves(x,y))
p_22_6(V1,V2,MT,MT2,RV0):-
  p_22_7(V1,MT,MT1,RV1), % tall(x)
  p_22_8(V1,V2,MT1,MT2,RV2), % loves(x,y) RV1 #/ RV2 #<=> RV0.
p_22_7(V1,MT,MT,RV0):-
  pred_var_at_typed(tall,[V1],MT,PV1),
  PV1 #= 1 #<=> RV0.
p_22_8(V1,V2,MT,MT,RV0):-
  pred_var_at_typed(loves,[V1,V2],MT,PV1),
  PV1 #= 1 #<=> RV0.
p_22_5(V1,MT,MT,RV0):-
  pred_var_at_typed(murderer,[V1],MT,PV1),
  PV1 #= 1 #<=> RV0.
SOLUTION:
GO!!
angry(mustard):1
angry(plum):1
angry(scarlett):1
angry(white):1
loves(mustard,scarlett):1
loves(plum,scarlett):1
loves(white,scarlett):1
murderer(white):1
person(green):1
person(mustard):1
person(peacock):1
person(plum):1
person(scarlett):1
person(white):1
short(green):1
short(mustard):1
short(plum):1
shortest(green):1
tall(peacock):1
tall(scarlett):1
tall(white):1
tallest(scarlett):1

```

Bibliography

- [1] Cyc Project Cycorp Inc. <http://www.cyc.com/research>, Feb 2013.
- [2] Cluedo game. http://www.hasbro.com/games/en_US/clue/, Jan 2014.
- [3] House M.D TV Series. <http://www.imdb.com/title/tt0412142>, Jan 2014.
- [4] Sherlock Holmes Stories. <http://www.sherlockholmes.com/>, Jan 2014.
- [5] Weka open source software repository. weka.sourceforge.net/doc/stable/weka/associations/Apriori.html, Jan 2014.
- [6] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington DC (USA)*, pages 207–216, 1993.
- [7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.
- [8] Plamen Angelov. *Autonomous Learning Systems: From Data Streams to Knowledge in Real-time*. John Wiley & Sons, 2012.
- [9] Paulo J Azevedo and Alípio M Jorge. Comparing rule measures for predictive association rules. In *Machine Learning: ECML 2007*, pages 510–517. Springer, 2007.
- [10] Leo Bachmair, Nachum Dershowitz, and David A Plaisted. Completion without failure. 1989.

- [11] Alan Baker. Simplicity. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2013 edition, 2013.
- [12] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.
- [13] Roman Barták. Constructive negation and constraints, 1998.
- [14] Roman Barták. Constructive negation in clp (h). Technical Report 98/6, Department of Theoretical Computer Science, Charles University, 1998.
- [15] Remco R Bouckaert, Eibe Frank, Mark Hall, Richard Kirkby, Peter Reutemann, Alex Seewald, and David Scuse. Weka manual for version 3-7-8. 2013.
- [16] Robert S Boyer, Matt Kaufmann, and J Strother Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995.
- [17] Ivan Bratko. *PROLOG Programming for Artificial Intelligence, Second Edition*. Addison-Wesley, 1990.
- [18] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 265–276, New York, NY, USA, 1997. ACM.
- [19] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 255–264, New York, NY, USA, 1997. ACM.
- [20] Alan Bundy, Simon Colton, and Toby Walsh. HR - a system for machine discovery in finite algebras. In *ECAI 98 Workshop Programme*. Wiley, 1998.

- [21] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.
- [22] Stanford University Center for Biomedical Informatics. Protégé Project. <http://protege.stanford.edu>, June 2014.
- [23] John Charnley and Simon Colton. Expressing general problems as CSPs. In *In Proceedings of the Workshop on Modelling and Solving Problems with Constraints at ECAI*, 2006.
- [24] John Charnley, Simon Colton, and Ian Miguel. Automated reformulation of constraint satisfaction problems. *Specification and Verification of Reconfiguration Protocols in Grid Component Systems*, page 8.
- [25] John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *ECAI*, pages 73–77, 2006.
- [26] John William Charnley. *A global workspace framework for combined reasoning*. PhD thesis, Imperial College London, 2009.
- [27] Jianzhong Chen, Stephen Muggleton, James Cussens, and Nicos Angelopoulos. Pepl: An implementation of fam for slps. *ALP Newsletter, focus on Probabilistic Prolog Systems*, 2011.
- [28] Jianzhong Chen, Stephen Muggleton, and José Santos. Learning probabilistic logic models from probabilistic examples. In *Proceedings of the 17th international conference on Inductive logic programming*, ILP’07, pages 22–23, Berlin, Heidelberg, 2008. Springer-Verlag.
- [29] S Colton. *Automated theory formation in pure mathematics*. Springer-Verlag, 2002.
- [30] S Colton and S Muggleton. Mathematical applications of Inductive Logic Programming. *Machine Learning*, 64:25–64, 2006.
- [31] Simon Colton. Hr - automatic concept formation in finite algebras. In *AAAI/IAAI*, page 1170, 1998.

- [32] Simon Colton, Alan Bundy, and Toby Walsh. Automatic concept formation in pure mathematics. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 786–791, 1999.
- [33] Simon Colton, Alan Bundy, and Toby Walsh. Automatic identification of mathematical concepts. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 183–190. Morgan Kaufmann Publishers Inc., 2000.
- [34] Simon Colton, Alan Bundy, and Toby Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human-Computer Studies*, 53(3):351–375, 2000.
- [35] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In *Principles and Practice of Constraint Programming CP 2001*, pages 575–579. Springer, 2001.
- [36] Simon Colton, Ramin Ramezani, and Maria Teresa Llano. The HR3 discovery system: Design decisions and implementation details. In *Proceedings of the AISB symposium on Computational Scientific Discovery*, 2014.
- [37] Simon Colton and Geraint A Wiggins. Computational creativity: The final frontier? 2012.
- [38] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system.
- [39] Robert Craven, Francesca Toni, Cristian Cadar, Adrian Hadad, and Matthew Williams. Efficient argumentation for medical decision-making. 2012.
- [40] James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.
- [41] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.
- [42] Luc De Raedt. *Logical and relational learning*. Springer, 2008.

- [43] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.
- [44] Luc De Raedt and Sašo Džeroski. First-order $\mathcal{I}_L$ $\mathcal{J}_K/\mathcal{I}_L$ -clausal theories are pac-learnable. *Artificial Intelligence*, 70(1):375–392, 1994.
- [45] Luc De Raedt and Kristian Kersting. Probabilistic logic learning. *ACM SIGKDD Explorations Newsletter*, 5(1):31–48, 2003.
- [46] Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In *Algorithmic Learning Theory*, pages 19–36. Springer, 2004.
- [47] Luc Dehaspe and H Toivonen. Frequent query discovery: a unifying ilp approach to association rule mining. *CW Reports*, page 33, 1998.
- [48] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In *Relational data mining*, pages 189–212. Springer, 2001.
- [49] Ana Luísa Duboc, Aline Paes, and Gerson Zaverucha. Using the bottom clause and mode declarations in fol theory revision from examples. *Machine learning*, 76(1):73–107, 2009.
- [50] Jesper Eskilson and Mats Carlsson. Sicstus mt - multithreaded execution environment for sicstus prolog. In *Implementation Technology for Programming Languages based on Logic*, pages 59–71, 1998.
- [51] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.
- [52] Jean H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.
- [53] Liqiang Geng and Howard J Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys (CSUR)*, 38(3):9, 2006.
- [54] Frank E Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11(1):1–21, 1969.

- [55] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.
- [56] Jiawei Han and Micheline Kamber. *Data Mining, Southeast Asia Edition: Concepts and Techniques*. Morgan kaufmann, 2006.
- [57] James Hawthorne. Inductive logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2014 edition, 2014. <http://plato.stanford.edu/entries/logic-inductive/>.
- [58] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. Waldmeister-high-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.
- [59] Anthony A Hilliard, Steven E Weinberger, Lawrence M Tierney Jr, David E Midthun, and Sanjay Saint. Occam’s razor versus saint’s triad. *New England Journal of Medicine*, 350(6):599–603, 2004.
- [60] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining: a general survey and comparison. *ACM sigkdd explorations newsletter*, 2(1):58–64, 2000.
- [61] John F Humphreys. *A course in group theory*, volume 6. Oxford University Press, 1996.
- [62] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679–688, 2006.
- [63] Hank Kahney. *Problem solving: A cognitive approach*. Open University Press Milton Keynes, 1986.
- [64] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*. Addison-Wesley Professional, 2014.
- [65] Janet L. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6(1):3–34, 1992.

- [66] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.
- [67] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3):227–260, 1972.
- [68] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.
- [69] George Lakoff and Rafael E Núñez. *Where mathematics comes from: How the embodied mind brings mathematics into being*. Basic books, 2000.
- [70] Stephane Lallich, Olivier Teytaud, and Elie Prudhomme. Association rule interestingness: measure and statistical validation. In *Quality measures in data mining*, pages 251–275. Springer, 2007.
- [71] James Larson and Ryszard S Michalski. Inductive inference of vl decision rules. *ACM SIGART Bulletin*, (63):38–44, 1977.
- [72] Nada Lavrač, Peter Flach, and Blaz Zupan. *Rule evaluation measures: A unifying view*. Springer, 1999.
- [73] David B Leake. Cbr in context: The present and furture. *Case-Based Reasoning Experiences, Lessons, & Future Directions*, pages 3–30.
- [74] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Fourth International Conference on Knowledge Discovery and Data Mining*, pages 80–86. AAAI Press, 1998.
- [75] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
- [76] Donald W Loveland. *Automated theorem proving: A logical basis (Fundamental studies in computer science)*. sole distributor for the USA and Canada, Elsevier North-Holland, 1978.

- [77] Jingyan Lu and Susanne P Lajoie. Supporting medical decision making with argumentation tools. *Contemporary Educational Psychology*, 33(3):425–442, 2008.
- [78] Richard E Mayer. *Thinking, problem solving, cognition* . WH Freeman/Times Books/Henry Holt & Co, 1992.
- [79] W McCune. The OTTER user’s guide. Technical Report ANL/90/9, Argonne National Laboratories, 1990.
- [80] William McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. 1994.
- [81] Ian Miguel. *Dynamic flexible constraint satisfaction and its application to AI planning*. PhD thesis, 2004.
- [82] Ian Miguel and Qiang Shen. Hard, flexible and dynamic constraint satisfaction. *The Knowledge Engineering Review*, 14(03):199–220, 1999.
- [83] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [84] S Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
- [85] S Muggleton. Stochastic logic programs. advances in inductive logic programming, ed. by luc de raedt, 1995.
- [86] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.
- [87] Stephen Muggleton. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.
- [88] Stephen Muggleton. Learning stochastic logic programs. *Computer and Information Science*, 5(41), 2000.

- [89] Stephen Muggleton. Learning structure and parameters of stochastic logic programs. In Stan Matwin and Claude Sammut, editors, *Inductive Logic Programming*, volume 2583 of *Lecture Notes in Computer Science*, pages 198–206. Springer Berlin Heidelberg, 2003.
- [90] Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In *Relational data mining*, pages 160–188. Springer, 2001.
- [91] Stephen Muggleton and Luc De Raedt. Inductive Logic Programming: Theory and Methods. *J. Log. Program.*, 19/20:629–679, 1994.
- [92] Stephen Muggleton, José Santos, and Alireza Tamaddoni-Nezhad. Progol: a system based on relative minimal generalisation. In *Inductive Logic Programming*, pages 131–148. Springer, 2010.
- [93] Shan-Hwei Nienhuys-Cheng and Ronald De Wolf. *Foundations of inductive logic programming*, volume 1228. Springer, 1997.
- [94] Karl Pearson. *On the theory of contingency and its relation to association and normal correlation; On the general theory of skew correlation and non-linear regression*. Cambridge University Press, 1904.
- [95] Alison Pease. A computational model of lakatos-style reasoning. 2007.
- [96] Alison Pease, Simon Colton, Ramin Ramezani, John Charnley, and Kate Reed. A discussion on serendipity in creative systems. In *Proceedings of the Fourth International Conference on Computational Creativity*, pages 64–71, 2013.
- [97] Alison Pease, Simon Colton, Ramin Ramezani, Alan Smaill, and Markus Guhe. Using analogical representations for mathematical concept formation. In *Model-Based Reasoning in Science and Technology*, pages 301–314. Springer Berlin Heidelberg, 2010.
- [98] Alison Pease, Simon Colton, Alan Smaill, and John Lee. A model of lakatos philosophy of mathematics. *Proceedings of Computing and Philosophy (ECAP)*, 2004.
- [99] Alison Pease, Alan Smaill, Simon Colton, Andrew Ireland, Maria Teresa Llano Rodriguez, Ramin Ramezani, Gudmund Grov, and Markus Guhe. *Applying Lakatos-style reasoning*

- to *AI domains*, pages 149–174. IGI Global, Thinking machines and the philosophy of computer science, 2010.
- [100] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of automated reasoning*, 2(2):191–216, 1986.
- [101] Gregory Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In *Knowledge Discovery in Databases*, pages 229–248. AAAI/MIT Press, 1991.
- [102] Karl R Popper. *Conjectures and Refutations*, volume 192. Basic Books New York, 1962.
- [103] Frederic Portoraro. Automated reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2014 edition, 2014. <http://plato.stanford.edu/entries/reasoning-automated/>.
- [104] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
- [105] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.
- [106] Ramin Ramezani and Simon Colton. Automatic generation of dynamic investigation problems. In *Automated Reasoning Workshop 2010 Bridging the Gap between Theory and Practice ARW 2010*.
- [107] Ramin Ramezani and Simon Colton. Solving mutilated problems. In *Automated Reasoning Workshop 2009 Bridging the Gap between Theory and Practice ARW 2009*, page 27.
- [108] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI communications*, 15(2):91–110, 2002.
- [109] Bradley L Richards and Raymond J Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
- [110] Sam Roberts. An introduction to Progol. *Department of Computer Science, University of York*, 1997.

- [111] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
- [112] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.
- [113] Jose Carlos Almeida Santos Santos. *Efficient learning and evaluation of complex concepts in inductive logic programming*. PhD thesis, Imperial College London, 2010.
- [114] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *IJCAI (1)*, pages 631–639, 1995.
- [115] Oliver Schulte. Formal learning theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014. <http://plato.stanford.edu/archives/spr2014/entries/learning-formal/>.
- [116] Ehud Y Shapiro. *Algorithmic program debugging*. MIT press, 1983.
- [117] Herbert A Simon and Allen Newell. Heuristic problem solving: The next advance in operations research. *Operations research*, 6(1):1–10, 1958.
- [118] Barbara Smith. *A tutorial on constraint programming*. 1995.
- [119] Paul Vincent Spade and Claude Panaccio. William of ockham. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2011 edition, 2011.
- [120] A Srinivasan. The aleph manual. university of oxford. <http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html>, 2007.
- [121] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
- [122] Geoff Sutcliffe and Christian Suttner. The tptp problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- [123] Alireza Tamaddoni-Nezhad and Stephen Muggleton. A note on refinement operators for ie-based ilp systems. In *ILP*, pages 297–314, 2008.

- [124] Alireza Tamaddoni-Nezhad and Stephen Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76(1):37–72, 2009.
- [125] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right objective measure for association analysis. *Information Systems*, 29(4):293–313, 2004.
- [126] Paul Thagard. Cognitive science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2012 edition, 2012. <http://plato.stanford.edu/archives/fall2012/entries/cognitive-science/>.
- [127] E Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.
- [128] Jean Van Heijenoort. *From Frege to Gödel: a source book in mathematical logic, 1879-1931*, volume 9. Harvard University Press, 1977.
- [129] Paul Vitanyi and Ming Li. Simplicity, information, kolmogorov complexity and prediction. *Zellner et al.(eds.)(2001)*, pages 135–55, 2001.
- [130] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.
- [131] Jan Wielemaker, S Ss, and I Ii. Swi-prolog 2.7-reference manual. 1996.
- [132] Chengqi Zhang and Shichao Zhang. *Association rule mining: models and algorithms*. Springer-Verlag, 2002.