Imperial College London
Department of Computing

Logic-based machine learning using a bounded hypothesis space: the lattice structure, refinement operators and a genetic algorithm approach

Alireza Tamaddoni Nezhad

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, August 2013

Abstract

Rich representation inherited from computational logic makes logic-based machine learning a competent method for application domains involving relational background knowledge and structured data. There is however a trade-off between the expressive power of the representation and the computational costs. Inductive Logic Programming (ILP) systems employ different kind of biases and heuristics to cope with the complexity of the search, which otherwise is intractable. Searching the hypothesis space bounded below by a bottom clause is the basis of several state-of-the-art ILP systems (e.g. Progol and Aleph). However, the structure of the search space and the properties of the refinement operators for theses systems have not been previously characterised. The contributions of this thesis can be summarised as follows: (i) characterising the properties, structure and morphisms of bounded subsumption lattice (ii) analysis of bounded refinement operators and stochastic refinement and (iii) implementation and empirical evaluation of stochastic search algorithms and in particular a Genetic Algorithm (GA) approach for bounded subsumption. In this thesis we introduce the concept of bounded subsumption and study the lattice and cover structure of bounded subsumption. We show the morphisms between the lattice of bounded subsumption, an atomic lattice and the lattice of partitions. We also show that ideal refinement operators exist for bounded subsumption and that, by contrast with general subsumption, efficient least and minimal generalisation operators can be designed for bounded subsumption. In this thesis we also show how refinement operators can be adapted for a stochastic search and give an analysis of refinement operators within the framework of stochastic refinement search. We also discuss genetic search for learning first-order clauses and describe a framework for genetic and stochastic refinement search for bounded subsumption. Finally, ILP algorithms and implementations which are based on this framework are described and evaluated.

Acknowledgments

First and foremost, I would like to thank my supervisor Stephen Muggleton for his generous support and inspiring guidance during all the time of my studies and research in his group.

I am grateful to the examiners Luc De Raedt and Marek Sergot for their careful reading and valuable suggestions which have improved this thesis. I would also like to thank Suresh Manandhar, Krysia Broda, Chris Bryant, Michele Sebag, Gerson Zaverucha, Flaviu Marginean, Philip Reiser and also the anonymous reviewers of the papers related to this thesis for their useful suggestions and comments.

I am also indebted to many students and researchers I had the pleasure to work with. I specially thank Jose Santos, Dianhuan Lin and Ghazal Milani with whom I had close collaborations which led to several joint papers.

I use this opportunity to thank my parents for their unconditional support. Last but not least, I would like to thank my wife, Parisa, and my daughter, Nikta, for their cheerful support and patience.

Declaration of Originality

I hereby declare that this thesis was composed by myself and that it describes my own research except where otherwise stated.

Alireza Tamaddoni Nezhad

Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Rich representation inherited from computational logic makes logic-based machine learning a competent method for knowledge-intensive problems, especially in application domains which involve relational background knowledge and structured data. Moreover, hypotheses represented in logic are directly comprehensible and the results of learning have the potential to be understandable by human experts. For example, this potential has been met in some applications of Inductive Logic Programming in computational and systems biology where the results have been meaningful to biologists and were sufficiently novel to be published in relevant scientific journals (see below).

Inductive Logic Programming (ILP) has been defined [Mug91] as the intersection of Machine Learning and Logic Programming. ILP is an inductive learning method which uses a logic-based representation and inference [MD94]. In other words, an ILP system develops first-order hypotheses from training examples and background knowledge which can all be represented as logic programs. ILP systems have been used successfully in a wide range of real-world applications (e.g. [DM92], [Moo97], [KDK97], [Moy02], [AD08] and [ARUK12]). In particular ILP has demonstrated remarkable success in challenging domains in computational and systems biology (e.g. [KMSS96] and [KWJ^{+}04]).

There is however a trade-off between the expressive power of the representation and the computational cost of reasoning within the representation [RN10]. In the case of learning in first-order logic, the computational cost is usually higher than other learning methods which use a less complex representation. Existing ILP systems mostly employ different kinds of biases and heuristics to cope with the complexity of the search, which otherwise is intractable. For example, using a syntactic bias restricts the set of hypotheses which are permitted.

Searching the hypothesis space bounded below by a bottom clause is the basis of several state-of-the-art ILP systems. In particular ILP systems such as Progol [Mug95] and Aleph [Sri07] are based on clause refinement through the hypothesis space bounded by a most specific clause which is constructed from background knowledge and a seed example using Inverse Entailment (IE) [Mug95]. These ILP systems use refinement operators together with a search method to explore a bounded hypothesis space in which each clause is guaranteed to cover at least one positive example (i.e. the seed example). It is known that the search space of these systems is limited to a sub-graph of the general subsumption lattice. However, the structure and properties of this search space have not been previously characterised.

In the first part of this thesis we characterise the hypothesis space considered by the ILP systems which use a bottom clause to constrain the search. We study the lattice, cover structure and morphisms of this bounded hypothesis space and give a new analysis of refinement operators for bounded subsumption.

Most ILP systems are traditionally based on clause refinement through a lattice defined by a generality order (e.g. subsumption). There is also a long-standing and increasing interest in stochastic search methods in ILP for searching the space of candidate clauses (e.g. [PKK93, Sri00, TNM02, RK03, ZSP06, PŽZ+07, MTN07, DPZ08]). However, to date there is very little theory to support the developments of these systems. In the second part of this thesis we try to answer the following question. How can the generality order of clauses and the relevant concepts such as refinement be adapted to be used in a stochastic search? To address this question we introduce the concept of stochastic refinement operators and adapt a framework, called stochastic refinement search. We also study a special case of stochastic refinement search where a genetic search algorithm and stochastic refinement operators are defined with respect to subsumption order relative to a bottom clause. Finally, we describe algorithms and implementations which are based on this framework and evaluate these on artificial and real-world problems.

This chapter briefly reviews the motivations, objectives and main contributions of this thesis. In the next section the background and motivations for this research, its significance and the rationale behind it are explained. Next the originality and contributions are reviewed and finally the organisation of the thesis is set out.

1.1 Background and motivations

1.1.1 Bounded hypothesis space

Machine learning can be characterised as a search problem [Mit97]. The complexity of the search is closely related to the complexity of the representation. In a typical ILP system the computational cost of the search depends on (i) the cost of evaluating clauses using subsumption testing which is known to be NP-complete [GJ79] and (ii) the size of the search space which can grow exponentially, e.g. with the maximum length of clauses. In order to make the search tractable, ILP systems use different kinds of biases. For example, a language bias restricts the set of acceptable hypotheses and a search bias specifies the way the system searches through the acceptable hypotheses. In ILP systems both language bias and search bias can be enforced by refinement operators. The concept of refinement and refinement operators are at the heart of many ILP systems and the refinement graph theory has been viewed as the main theoretical foundation of ILP [NCdW97]. Different properties of refinement operators such as properness, completeness and idealness have been studied in the literature and the structure and the completeness of the search can be determined based on these properties.

ILP systems such as Progol [Mug95] and Aleph [Sri07] are based on clause refinement through the hypothesis space bounded by a bottom clause which is constructed from background knowledge and a seed example using Inverse Entailment (IE) [Mug95]. In this setting, refinement operators are used together with a search method to explore a bounded hypothesis space in which each clause is guaranteed to cover at least one positive example (i.e. the seed example). These systems have been successfully applied to a wide range of real-world application, however, the structure of the search space and the properties of the refinement operators for these systems have not been previously characterised. In this thesis we introduce bounded subsumption in order to characterise the hypothesis space considered by these ILP systems. We also study the lattice structure and refinement operators for the bounded subsumption.

1.1.2 Stochastic search

Despite a bounded refinement and single target clause assumption, the size of the search space in a Progol-like ILP system can still grow exponentially with the size of the target clause [Mug95, Sri00]. However, the complexity of the search is less evident when dealing with small clauses. It is therefore not surprising to note that most successful applications of ILP systems described in the current literature involve clauses with only a small number of literals and variables [1]. For example most ILP systems which have been applied to the mutagenesis and carcinogenesis problems [KMSS96, SMS97] have used a syntax bias to restrict the search to clauses involving only few literals (i.e. up to 4 literals). In some other applications of ILP which involve clauses with a larger number of literals, domain specific information has been used to prune the search. For example an acceptable description of a pharmacophore can be specified explicitly by the user and can be used by the ILP system as a declarative bias to avoid searching for pharmacophores which are not acceptable [FMPS98]. However, for many real-world applications this kind of domain specific knowledge about the acceptable forms of the target concept is not known beforehand. On the other hand, ILP systems will need to scale up to deal with more complex target concepts and larger datasets. Moreover, studies [GS00, RKD02, ZSP03] suggest that some phenomena such as phase transition [HHW96] and heavy-tailed distribution [GSCK00] which have been observed for satisfiability problems also exist in some relational learning problems including real-world applications such as mutagenesis. Some of these studies [BGSS03] also suggest that existing techniques used in most relational learning systems cannot easily scale up for learning concepts more complex than those described in the current literature.

Existing ILP systems mostly employ downward or upward refinement operators and

[1] Exceptions include Golem [MF90] (e.g [MKS92]) and ProGolem [MSTN10a] (e.g [SNCDP12]). Both systems use lgg-like operators. Golem is limited to determinate clauses while ProGolem does not have this limitation as discussed in Chapter 7.

deterministic search methods to explore a lattice of clauses ordered by subsumption [NCdW97]. Depending on the direction of the search, ILP systems are classified as either top-down (general to specific) or bottom-up (specific to general). For similar complex problems in other branches of Artificial Intelligence (AI), there has been an increasing interest in using stochastic instead of deterministic search methods. For example, for the Satisfiability Problems (SAT) and the Travelling Salesperson Problem (TSP), which are well known computationally complex problems, stochastic algorithms have outperformed other search methods which are known for these problems. Examples of stochastic search methods which have been successfully applied in these problems are Simulated Annealing [Sam99, Ghe94], Iterated Local Search (e.g. GSAT and WalkSAT) [SKC94, SLM92], Genetic Algorithms [JS89, BP00, FPS01, PW02], and Ant Colony Optimisation [GMS01, BGD02, SN00].

Stochastic and probabilistic search methods have also been used in ILP both for the clause evaluation task [SR00, Sri99] and for searching the space of candidate clauses [Sri00, PS03, ZSP03]. Nevertheless, designing novel search methods has been identified as a pressing issue for ILP and relational learning [PS03, Sri05, SGC11].

1.1.3 Stochastic refinement and genetic search

In this thesis we also try to answer the following question. How can the generality order of clauses and the relevant concepts such as refinement be adapted to be used in a stochastic search? To address this question we introduce the concept of stochastic refinement operators and adapt a framework, called stochastic refinement search.

We also study a special case of stochastic refinement search where unary and binary stochastic refinement operators and a Genetic Algorithm (GA) search are defined with respect to bounded subsumption. GAs are stochastic yet highly directed search methods which are based on a mechanism inspired by natural evolution [Hol75, Gol89]. The idea of using genetic and evolutionary algorithms in machine learning can be traced back to the early days of computer science and artificial intelligence:

> *'It is probably wise to include a random element in a learning machine ... Since there is probably a very large number of satisfactory solutions the ran-*

dom method seems to be better than the systematic. It should be noticed that it is used in the analogous process of evolution'
Alan M. Turing [Tur50]

One important feature of GAs which makes them different from other search methods is that they tend to rely on recombination (also known as crossover) as the principal search mechanism. GAs combine survival of the fittest among the problem solutions with a structured information exchange to form a search algorithm which has some of the innovative flair of human search [Gol02]. The recombination operator is especially useful in many problems where the solutions feature recombinable building blocks (i.e. partial solutions). This also gives more exploration power to the search in order to work as a global strategy compared to other stochastic search methods which use local strategies that step only to neighboring states.

The recombination operator in GAs is relevant to binary refinement in ILP. It has been shown [MM99] that the minimum depth of any clause within the binary refinement graph is logarithmically related to the depth in the refinement graph of corresponding unary operator whenever a certain lattice divisibility assumption is met.

1.2 Originality and contributions

It is known that the search space of a Progol-like ILP system is limited to a sub-graph of the general subsumption lattice. Progol's refinement operator and its incompleteness with respect to the general subsumption order were initially discussed in [Mug95] and [BS99]. A new subsumption order was also suggested [BS99] for characterising Progol's refinement, which as we show in this thesis, cannot capture all aspects of Progol's refinement. In this thesis we characterise the hypothesis space considered by the ILP systems which use a bottom clause to constrain the search. In particular we discuss and characterise refinement in Progol as a representative of these ILP systems. We study the lattice and cover structure of this bounded hypothesis space and give a new analysis of refinement operators, least generalisation and greatest specialisation in the subsumption order relative to a bottom clause. We also show the morphisms between the lattice of bounded subsumption, an atomic lattice and the lattice of partitions.

This analysis is important for better understanding the constrained refinement space of ILP systems such as Progol and Aleph which proved to be successful for solving real-world problems (despite being incomplete with respect to general subsumption order). Moreover, characterising this refinement sub-lattice can lead to more efficient ILP algorithms and operators. For example, we show that, by contrast with the general subsumption order, efficient least and minimal generalisation operators can be designed for the subsumption order relative to a bottom clause. These efficient operators are the basis of the ILP systems GA-Progol [TNM02, MTN07] and ProGolem [MSTN10a]. The theoretical results about the bounded refinement and refinement operators relative to a bottom clause are applicable to ILP systems such as Progol and Aleph which use some form of Inverse Entailment (IE). Moreover, these results are also applicable to other ILP systems which use a bottom clause to restrict the search space. These include ILP systems which use stochastic algorithms to explore the hypothesis space bounded by a bottom clause (e.g. [Sri00, ZSP03, PŽZ+07, DPZ08]). The search space of these systems can be characterised by bounded subsumption described in this thesis.

The refinement graph theory has been viewed as the main theoretical foundation of ILP [NCdW97]. Since the publication of this theory, there have been attempts to build ILP systems based on stochastic and randomised methods. However, to date there is very little theory to support the developments of these systems. In this thesis we discuss how the refinement theory and relevant concepts such as refinement operators can be adapted for a stochastic ILP search. Stochastic refinement is introduced as a probability distribution over a set of clauses and can be viewed as a prior in a stochastic ILP search. We study the properties of a stochastic refinement search as two well known Markovian approaches: 1) Gibbs sampling algorithm [Mit97] and 2) Random heuristic search [Vos99]. We also discuss genetic search for learning first-order clauses and describe a framework for genetic and stochastic refinement search for bounded subsumption. ILP algorithms implemented based on this framework are evaluated and discussed in this thesis.

The main contributions of this thesis can be summarised as follows:

1. We characterise clause refinement and the search space of the ILP systems which use a bounded hypothesis space. For this purpose, we introduce the concept of

bounded subsumption, i.e. subsumption relative to a bottom clause (Chapter 3).

2. We study the lattice and cover structure of bounded subsumption. We also show the morphisms between the lattice of bounded subsumption, an atomic lattice and the lattice of partitions. We show that ideal refinement operators exist for bounded subsumption. We also show that, by contrast with the general subsumption order, efficient least generalisation operators can be designed for the bounded subsumption (Chapters 4 and 5).

3. Stochastic refinement is introduced to show how refinement operators can be adapted for a stochastic ILP search. We give an analysis of stochastic refinement operators within the framework of stochastic refinement search. This can be used to characterise stochastic search methods in some ILP systems. We discuss genetic search for learning first-order clauses and describe a special case of stochastic refinement search where stochastic refinement operators and a GA search are defined with respect to bounded subsumption (Chapters 6).

4. Implementation and evaluation of novel stochastic refinement operators and a genetic search algorithm for the bounded subsumption are discussed in this thesis. ILP algorithms implemented based on the framework of stochastic refinement relative to bounded subsumption are discussed and evaluated on artificial and real-world problems. In particular we describe a genetic algorithm approach which uses the encodings and operators for the bounded subsumption and can also be characterised as a stochastic refinement search. (Chapters 7 and 8).

1.3 Organisation of the thesis

In the following we briefly review the contents of each chapter.

• Chapter 1, **Introduction.**

This chapter presents an overview of the thesis and reviews the motivations, objectives and main contributions of this project.

• Chapter 2, **Machine learning and Inductive Logic Programming (ILP).**

The purpose of this chapter is to provide the background information and key concepts which are needed in the next chapters. We give an introduction to ILP in the context of Machine Learning and describe the ILP problem setting and important elements such as subsumption and refinement operators.

• Chapter 3, **Hypothesis space and bounded subsumption.**
In this chapter we review and discuss clause refinement in Progol as a representative of ILP systems which are based on clause refinement through the hypothesis space bounded by a most specific clause. We introduce a subsumption order relative to a bottom clause and demonstrate how Progol's refinement can be characterised with respect to this order.

• Chapter 4, **The lattice structure and morphism of bounded subsumption.**
In this chapter we study the lattice and cover structure of the bounded subsumption and show the morphism between this lattice, an atomic lattice and the lattice of partitions.

• Chapter 5, **Encodings and refinement operators for bounded subsumption.**
Encodings and refinement operators for the bounded subsumption are discussed in this chapter. We study the properties of refinement operators in the subsumption order relative to a bottom clause and show that ideal refinement operators exist for the bounded subsumption.

• Chapter 6, **Stochastic refinement and genetic search.**
In this chapter we study how refinement operators can be adapted for a stochastic ILP search. Stochastic refinement is introduced and we give an analysis of stochastic refinement operators within the framework of stochastic refinement search. We also study a special case of stochastic refinement search where genetic refinement operators are defined with respect to subsumption order relative to a bottom clause.

• Chapter 7, **Algorithms and implementation.**
In this chapter we describe algorithms and implementation which are based on the framework of stochastic refinement relative to bottom clause. In particular we discuss stochastic refinement searches implemented in systems GA-Progol and ProGolem.

• Chapter 8, **Empirical evaluation.**
The algorithms and implementations described in the previous chapter are evaluated

on artificial and real-world problems and we study and compare their performances.

• Chapter 9, **Conclusions.**
This chapter summarises the main contributions and concludes the thesis.

1.4 Publications

Parts of this thesis have already appeared in print elsewhere:

• Parts of Chapters 3, 4 and 5 have appeared in Proceedings of the 18th International Conference on Inductive Logic Programming [TNM08] and the Machine Learning Journal [TNM09].

• Parts of Chapter 6 have appeared in the Proceedings of the 20th International Conference on Inductive Logic Programming [TNM11].

• Parts of Chapters 7 and 8 have appeared in the Proceedings of the 10th, 12th and 19th International Conference on Inductive Logic Programming [TNM00], [TNM02] and [MSTN10a] [2], Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) [TNM01] and the Machine Learning Journal [MTN07] [3].

• Parts of Chapter 9 have appeared in the Proceedings of the 21th and 23rd International Conference on Inductive Logic Programming [TNBRM12], [MLCTN13], the PLoS ONE Journal [BCLM$^+$11], Advances in Ecological Research Journal [TNMR$^+$13] and the Machine Learning Journal [MLPTN14].

The author of this thesis has other publications which are mainly on real-world applications of the ILP systems described in this thesis (e.g. [MTNW03], [TNKMP04], [TNCKM06], [TNCK$^+$07] and [STNL$^+$13]) or development of relevant ILP systems (e.g. [MSTN10b] and [MLTN12]) and therefore are indirectly related to the scope of this thesis.

[2] The author contributed to the theoretical framework and the design of ARMGs operators.

[3] The author's contribution to this paper was the design and implementation of QG/GA and performing the experiments. The idea and initial implementation of QG was due to Stephen Muggleton.

Chapter 2

Machine learning and Inductive Logic Programming (ILP)

Machine learning is a branch of Artificial Intelligence (AI) concerned with the design and development of computer programs which can improve their performance in a given task with the input information (e.g. examples, descriptions, sensory data) [Mit97]. A major application of machine learning has been to automatically find useful patterns from empirical data. As in human learning, in a machine learning problem we may already have some prior or background knowledge, relevant to the learning task, which can facilitate the learning. Kodratoff and Michalski [KM90] provided a multi-criterion classification of machine learning methods which is still applicable to most machine learning approaches today. According to this classification, machine learning methods can be characterised based on different criteria such as the primary purpose of the learning (e.g. synthetic, analytic), type of input (e.g. examples, observation), type of primary inference (e.g. inductive, deductive) and the role of prior knowledge (e.g. empirical, axiomatic). Synthetic learning, which uses induction as the primary inference, aims primarily at creating new or better knowledge, and analytic learning, which uses deduction as the primary inference, aims primarily at reformulating given knowledge into a better form.

Induction can be viewed as reversing the process of deduction. While deduction is a derivation of consequents from given premises, induction is a process of hypothesising premises that entail given consequents. Inductive learning involves finding a hypothesis which entails positive examples and does not entail any of the negative examples.

This can be achieved by empirical generalisation of positive examples with or without using background knowledge. In deductive learning methods, such as explanation-based learning [MKKC86], the background knowledge already entails all the positive examples, however, it requires reformulating, e.g. to make sure it is in a tractable form. Induction is the fundamental process used by human beings for formulating general laws from specific observations. Induction as a form of reasoning was first described in Aristotle's Posterior Analytics [Bar75] and the importance of inductive inference as a source of scientific knowledge has been studied by other philosophers such as F. Bacon [Bac20] and C. Peirce [Pei32].

As in other branches of AI, knowledge representation is an important aspect in machine learning. Logic was recognised as a key representation method since the early days of artificial intelligence [Tur50, McC59]. The development of computational logic in the form of logic programming [Kow80] has played an important role in advancing different areas of AI, including machine learning. Hence, logic-based machine learning has been mainly concerned with Horn clauses (standard representation in logic programming) despite the fact that some theoretical foundations (e.g. [Plo71]) are not limited to Horn clauses. This trend has continued to date, where logic-based machine learning is regarded [1] by the majority to be synonymous with machine learning of logic programs, also known as Inductive Logic Programming. Inductive Logic Programming (ILP), defined [Mug91] as the intersection of Machine Learning and Logic Programming, is an inductive learning method which uses a logic-based representation (in the form of logic programs) and inference [MD94]. ILP has its roots in the theoretical foundations developed by Plotkin [Plo71], Reynolds [Rey69] and Vere [Ver75] and the development of the systems MIS [Sha83], MARVIN [SB86] and CIGOL [MB88]. However, the term Inductive Logic Programming was first used by Muggleton in his seminal paper [Mug91] and the same year he organised the first of international workshops on ILP which have been held annually since then.

In addition to the ILP workshop and conference proceedings, several books have been published on the theoretical foundations and methodologies of ILP, e.g. [Mug92], [LD93], [De 96], [NCdW97] and [De 08]. There are also several papers which survey the

[1] This is also the view which we consider in this thesis, even though some of the results could be extended to non-Horn clause representation.

field, e.g. [PS03], [Sri05] and [MRP$^+$11]. Many ILP systems, implementing different ILP methodologies, have been developed. These include FOIL [Qui90], Golem [MF90], CLINT [DB91], LINUS [LDG91], Progol [Mug95], CLAUDIEN [DD97], TILDE [BD97], WARMR [DT99], TopLog [MSTN10b] and ProGolem [MSTN10a].

ILP systems have been used successfully in a wide range of real-world applications including natural language processing [ZM93, Moo97, Cus97, DCM00, SSJ$^+$09], mechanical engineering [DM92, Kar95, ŽŽGS$^+$07], ecology [DDRW94, KDK97, TNMR$^+$13], robot navigation [Kli94, Moy02, CKP$^+$06], software engineering [BG95, CD97, BV09, ARUK12], music [Dov95, VBD97, AD08] and many more. In particular ILP has demonstrated remarkable success in challenging domains in computational biology including predictive toxicology [KMSS96, SMKS96, SMS97, ASHMS07], pharmacophore design [FMPS98, SM03, SPCK06], protein structure prediction [KS90, MKS92, TMS98, CMS03, LMS10] and systems biology [KWJ$^+$04, TNCKM06, SDI09, LM10].

There are also several multi-disciplinary developments in ILP including Probabilistic ILP (e.g. SLPs [Mug96], BLPs [KD01], PRISM [Sat05], MLN [DKP$^+$06] and ProbLog [DKT07]) and Abductive ILP (e.g. [MB00], [TNCKM06] and [IFN10]).

This chapter provides the background information and key concepts from ILP which are needed in the next chapters. As preliminaries we review definitions from clausal logic as well as some basic definitions from ordered sets and lattice theory. In this chapter we describe the ILP problem setting and important elements such as subsumption and refinement operators. Then we briefly discuss the ILP systems Golem and Progol which are important in the context of this thesis.

2.1 Preliminaries

For the sake of completeness and clarity of notation, in this section we present definitions from clausal logic as well as some basic definitions from ordered sets and lattice theory.

2.1.1 Definitions from clausal logic

This section is intended as a brief reminder of the concepts from clausal logic which are used in this thesis. The definitions in this section are mostly adapted from [NCdW97] and [Mug95].

Syntactical structures in clausal logic are constructed from a set of symbols known as clausal alphabet.

Definition 1 (Clausal alphabet) *A clausal alphabet consists of the following symbols:*

- *variables represented by an upper case letter followed by a string of lower case letters and digits.*
- *function symbols represented by a lower case letter followed by a string of lower case letters and digits.*
- *predicate symbols represented by a lower case letter followed by a string of lower case letters and digits.*
- *connectives including* $\vee$, $\wedge$, $\neg$ *and* $\leftarrow$
- *quantifiers* $\forall$ *and* $\exists$
- *punctuation symbols including "(", ")" and ","*

Definition 2 (Term) *A term is defined recursively as follows:*

- *a variable is a term.*
- *if f is an $n-$ary function symbol ($n \geq 0$) and $t_1, t_2, \ldots, t_n$ are terms then $f(t_1, t_2, \ldots, t_n)$ is a term.*

A term $a()$ where a is a $0-$ary function symbol is called a constant and is normally denoted by a. '[]' and '.' are function symbols and if $t_1, t_2, \ldots$ are terms then '.'(t_1, t_2) can equivalently be denoted $[t_1|t_2]$ and '.'$(t_1,$'.'$(t_2, ..$'.'$(t_n, []) ..))$ can equivalently be denoted $[t_1, t_2, .., t_n]$.

Using terms, formulas can be constructed and the smallest possible formula is called *atom*.

Definition 3 (Atom) *If p is an $n-$ary predicate symbol and $t_1, t_2, \dots, t_n$ are terms then $p(t_1, t_2, \dots, t_n)$ is called an atomic formula, or atom.*

An atom $p()$ where p is a $0-$ary predicate symbol is normally denoted by p.

Definition 4 (Well-formed formula (wff)) *Every atom is a well-formed formula (wff). If W and W' are wffs then $\neg W$ (not W), $W \wedge W'$ (W and W'), $W \vee W'$ (W or W') and $W \leftarrow W'$ (W implied by W') are wffs. $W \wedge W'$ is a conjunction and $W \vee W'$ is a disjunction. If v is a variable and W is a wff then $\forall v.W$ (for all v W) and $\exists v.W$ (there exists a v such that W) are wffs. v is said to be universally quantified in $\forall v.W$ and existentially quantified in $\exists v.W$.*

The wff W is said to be function-free if and only if W contains no function symbols. $\text{vars}(W)$ denotes the set of variables in W. W is said to be ground if and only if $\text{vars}(W) = \emptyset$.

The basic building blocks in clausal normal form are literals, which in turn make up clauses.

Definition 5 (Literal) *A literal is an atom A or the negation $\neg A$ of an atom A. A is called a positive literal and $\neg A$ is called a negative literal.*

Definition 6 (Clause) *A clause is a finite set of literals. The empty clause, denoted by $\Box$, is the clause containing no literals. A clause represents the disjunction of its literals. Thus the clause $\{a_1, a_2, ..\neg a_i, \neg a_{i+1}, .., \neg a_n\}$ can be equivalently represented as $(a_1 \vee a_2 \vee ..\neg a_i \vee \neg a_{i+1} \vee .. \vee \neg a_n)$ or $a_1; a_2; .. \leftarrow a_i, a_{i+1}, .., a_n$. All the variables in a clause are implicitly universally quantified. A Horn clause is a clause which contains at most one positive literal. A definite clause is a clause which contains exactly one positive literal. A positive literal in either a Horn clause or definite clause is called the head of the clause while the negative literals are collectively called the body of the clause.*

Definition 7 (Logic program) *A logic program is a set of definite clauses.*

Definition 8 (Clausal theory) *A set of clauses in which no pair of clauses share a common variable is called a clausal theory. A clausal theory represents the conjunction of its clauses. Thus the clausal theory* $\{C_1, C_2, .., C_n\}$ *can be equivalently represented as* $(C_1 \wedge C_2 \wedge .. \wedge C_n)$.

Every clausal theory is said to be in clausal normal form. Every wff can be transformed to a wff in clausal normal form. If $C = \forall l_1 \vee ..l_n$ is a clause then $\neg C = \exists \neg l_1 \wedge .. \wedge \neg l_n$. In this case $\neg C$ is not in clausal normal form since the variables are existentially quantified. $\neg C$ can be put in clausal normal form by substituting each occurrence of every variable in $\neg C$ by a unique constant not found in C. This is a special case of a process known as Skolemisation and the unique constants are called Skolem constants. Note that the Skolemised form of a wff is not equivalent to the original wff but the original wff is unsatisfiable if and only if Skolemised form is unsatisfiable.

A substitution replaces variables in a formula by terms.

Definition 9 (Substitution) *A substitution* θ *is a set* $\{v_1/t_1, .., v_n/t_n\}$ *where each* v_i *is a distinct variable and each* t_i *is a term. We say* t_i *is substituted for* v_i *and* v_i/t_i *is called a binding for* v_i. *The set* $\{v_1, .., v_n\}$ *is called the domain of* θ, *or dom*(θ), *and* $\{t_1, .., t_n\}$ *the range of* θ, *or rng*(θ). *A substitution* $\theta = \{v_1/t_1, .., v_n/t_n\}$ *is called a variable substitution if every* t_i *is a variable. A variable substitution* $\theta = \{u_1/v_1, .., u_n/v_n\}$ *is said to be a variable renaming if and only if dom*(θ) *is disjoint from rng*(θ) *and each* v_i *is distinct.*

A substitution can be applied to a formula by replacing variables in the formula by terms according to the substitution. This is called instantiation.

Definition 10 (Instantiation) *Let* E *be a wff or a term and* $\theta = \{v_1/t_1, .., v_n/t_n\}$ *be a substitution. The instantiation of* E *by* θ, *written* $E\theta$, *is formed by replacing every occurrence of* v_i *in* E *by* t_i.

Sometimes we need to rename the variables in a formula. In this case, the new formula which is equivalent to the old one is called a variant of the old formula. Such a variant can be obtained by applying a renaming substitution.

Definition 11 (Alphabetic variants) *Let W and W' be two wffs, W and W' are said to be alphabetic variants of each other if there exists a variable renaming θ such that $W\theta = W'$. Wffs W, W' are said to be standardised apart if and only if there exists a variable renaming $\theta = \{u_1/v_1, ..u_n/v_n\}$, vars$(W) \subseteq$ vars(θ) and $W\theta = W'$.*

A set of formulas is unifiable if there exists a substitution which if applied to them will make them identical. This substitution is called a unifier.

Definition 12 (Unifier and most general unifier (mgu)) *The substitution θ is said to be the unifier of the atoms a and a' whenever $a\theta = a'\theta$. μ is the most general unifier (mgu) of a and a' if and only if for all unifiers γ of a and a' there exists a substitution δ such that $(a\mu)\delta = a\gamma$.*

The semantics of a formula in a language is concerned with the meaning attached to the formula. Herbrand interpretations, named after the French logician Jacques Herbrand, are important in clausal logic. First we define the Herbrand universe (the set of all ground terms in the language) and the Herbrand base (the set of all ground atoms in the language).

Definition 13 (Herbrand universe) *The Herbrand universe of the wff W is the set of all ground terms composed of constants and function symbols found in W. If W does not contain any constant then we add an arbitrary constant to the alphabet to be able to compose ground terms.*

Definition 14 (Herbrand base) *The Herbrand base of the wff W is the set of all ground atoms composed of predicate symbols found in W and the terms in the Herbrand universe of W.*

Definition 15 (Herbrand interpretation) *A Herbrand interpretation I of wff W is a total function from ground atoms in the Herbrand base of W to $\{\square, \blacksquare\}$, where symbols $\square$ and $\blacksquare$ represent [2] the logical truth values False and True respectively.*

A Herbrand interpretation I of wff W can also be represented as the subset of the atoms

[2] Note that the symbols $\square$ and $\blacksquare$ also represent the empty clause and the empty theory respectively.

a in the Herbrand base of W for which $I(a) = \blacksquare$. In the following, all interpretations I are assumed to be Herbrand.

- The atom a is true in I if $I(a) = \blacksquare$ and false otherwise.
- The wff $\neg W$ is true in I if W is false in I and is false otherwise.
- The wff $W \wedge W'$ is true in I if both W and W' are true in I and false otherwise.
- The wff $W \vee W'$ is true in I if either W or W' is true in I and false otherwise.
- The wff $W \leftarrow W'$ is true in I if $W \vee \neg W'$ is true in I and false otherwise.
- If v is a variable and W is a wff then $\forall v.W$ is true in I if for every term t in the Herbrand universe of W the wff $W\{v/t\}$ is true in I. Otherwise $\forall v.W$ is false in I.
- If v is a variable and W is a wff then $\exists v.W$ is true in I if $\neg\forall v.\neg W$ is true in I and false otherwise.

An interpretation which makes a formula true is called a model for that formula.

Definition 16 (Model) *Interpretation M is a model of wff W if and only if W is true in M.*

Definition 17 (Semantic entailment) *Let W and W' be two wffs. We say that W semantically entails W', or $W \models W'$ if and only if every model of W is a model of W'.*

Definition 18 (Satisfiable) *A wff W is satisfiable if there exists a model of W and unsatisfiable otherwise. Consequently W is unsatisfiable if and only if $W \models \square$.*

It is shown that a formula is satisfiable if and only if it has a Herbrand model. This means that in testing unsatisfiability of a formula we can restrict attention to Herbrand models.

Proposition 1 *The wff W is satisfiable if and only if W has a Herbrand model.*

Proposition 2 *Every logic program P has a unique least Herbrand model M such that M is a model of P and every atom a is true in M only if it is true in all Herbrand models of P.*

Proposition 3 *Let X, Y and Z be wffs. Then $X \wedge Y \models Z$ if and only if $X \models \neg Y \vee Z$.*

In clausal logic, an inference rule can be viewed as a function which takes premises, analyses their syntax, and draws a conclusion from the premises.

Definition 19 (Inference rule) *An inference rule, denoted by $\frac{X}{Y}$, is a rule whereby one may draw a conclusion Y from one or more premises X.*

Definition 20 (Sound inference) *Let $\frac{X}{Y}$ be an inference rule. Then $\frac{X}{Y}$ is said to be sound if and only if $X \models Y$.*

Syntactic entailment can be defined based on inference rules.

Definition 21 (Syntactic entailment) *Suppose I is a set of inference rules containing $\frac{X}{Y}$ and W, W' are wffs. Then $W \vdash_I W'$ if W' is formed by replacing an occurrence of X in W by Y. Otherwise $W \vdash_I W'$ if $W \vdash_I W''$ and $W'' \vdash_I W'$. We say that W syntactically entails W' using inference rules I, if and only if $W \vdash_I W'$. The set of inference rules I is said to be deductively sound and complete if and only if each rule in I is sound and $W \vdash_I W'$ whenever $W \models W'$.*

Definition 22 (Entailment generality order) *Let W and W' be two wffs. We say that W is more general than W' (conversely W' is more specific than W) if and only if $W \models W'$.*

Apart from the entailment generality order, there are other generality orders between logical clauses. One important generality order which is widely used in ILP is the subsumption order [Plo71]. The subsumption order and lattice are defined in Section 2.3. In the following section we first review some definitions from orders and lattices.

2.1.2 Orders and lattices

In this section we present some basic definitions from ordered sets and lattice theory. These definitions are adapted from the literature (e.g. [DP02]) to suit the needs of this thesis.

Definition 23 (Relations) *Let P be a set and R be a binary relation on P. The relation R is said to be:*

- *a* reflexive relation *if for all a in P we have aRa.*
- *a* transitive relation *if for all a, b and c in P, if aRb and bRc then aRc .*
- *a* symmetric relation *if for all a and b in P, if aRb then bRa .*
- *an* antisymmetric relation *if for all a and b in P, if aRb and bRa then $a = b$.*
- *a* total relation *if for all a and b in P, aRb or bRa.*
- *an* equivalence relation *if R is reflexive, transitive and symmetric.*

Definition 24 (Functions) *Let P and Q be sets and $f \subseteq P \times Q$ be a binary relation. The relation f is said to be:*

- *a* function *or* mapping *from P to Q, denoted by $f : P \to Q$, if $\forall x \in P$ the set $\{y \in Q | (x, y) \in f\}$ has exactly one element. In this case domain and range of f are defined as follows. $D(f) = P$ and $R(f) = \{y | x \in D(f), (x, y) \in f\}$.*
- *a* surjective *or* onto function *if f is a function where $\forall y \in Q$ there exists $x \in P$ such that $f(x) = y$.*
- *a* injective *or* one-to-one function *if f is a function where $\forall x_1, x_2 \in P$, $f(x_1) = f(x_2)$ implies $x_1 = x_2$.*
- *a* bijective function *if f is a function which is both* surjective *and* injective.
- *a* monotonically increasing function *if f is a function where $\forall x_1, x_2 \in P$, $x_1 < x_2$ implies $f(x_1) \leq f(x_2)$.*

- *a* strictly increasing function *if f is a function where $\forall x_1, x_2 \in P$, $x_1 < x_2$ implies $f(x_1) < f(x_2)$.*

Definition 25 (Ordered sets) *Let P be a set and R be a binary relation on P. The pair $\langle P, R\rangle$ is said to be:*

- *a* quasi-ordered set *if R is reflexive and transitive.*
- *a* partially ordered set *if R is reflexive, transitive and antisymmetric.*
- *a* totally ordered set *if R is total, transitive and antisymmetric.*

Definition 26 (Mappings between ordered sets) *Let $\langle P, \leq\rangle$ and $\langle Q, \subseteq\rangle$ be quasi-ordered sets. A mapping $f : P \to Q$ is said to be:*

- order-preserving *(or* monotone*) if for all x and y in P, $x \leq y$ implies $f(x) \subseteq f(y)$.*
- order-embedding *if for all x and y in P, $x \leq y$ if and only if $f(x) \subseteq f(y)$.*
- order-isomorphism *if it is an order-embedding which maps P onto Q. In this case we say P and Q are (order) isomorphic and write $P \cong Q$.*

Proposition 4 *Let $\langle P, \leq\rangle$ and $\langle Q, \subseteq\rangle$ be partially ordered sets and $f : P \to Q$ be an order-isomorphism. Then we have $x = y$ if and only if $f(x) = f(y)$*

Proposition 5 *Let $\langle P, \leq\rangle$ and $\langle Q, \subseteq\rangle$ be partially ordered sets and $f : P \to Q$ be an order-isomorphism. Then the inverse of f, $f^{-1} : Q \to P$, is also an order-isomorphism.*

Definition 27 (lub and glb) *Let $\langle P, \leq\rangle$ be a quasi-ordered set and $S \subseteq P$. An element $x \in P$ is an* upper bound *of S if $s \leq x$ for all $s \in S$. An upper bound x of S is a* least upper bound (lub) *of S if $x \leq z$ for all upper bounds z of S. Dually, an element $x \in P$ is a* lower bound *of S if $x \leq s$ for all $s \in S$. A lower bound x of S is a* greatest lower bound (glb) *of S if $z \leq x$ for all lower bounds z of S.*

Definition 28 (Lattice) *A quasi-ordered set $\langle L, \leq\rangle$ is called a lattice if for every x and y in L a lub of $\{x, y\}$, also denoted by $x \vee y$ (read 'x join y') and a glb of $\{x, y\}$,* *also denoted by $x \wedge y$ (read 'x meet y') exist. A lattice $\langle L, \leq \rangle$ is also denoted using the meet and join operators: $\langle L, \wedge, \vee \rangle$.*

Upward and downward covers for a quasi-ordered set can be viewed as the smallest possible non-trivial upward or downward steps in the quasi-order.

Definition 29 (Upward/downward covers) *Let $\langle L, \leq \rangle$ be a quasi-ordered set and $x, y \in L$. If $x < y$ and there is no $z \in L$ such that $x < z < y$, then x is an upward cover of y, and y is a downward cover of x.*

Definition 30 (Ascending/descending chain) *Let $\langle L, \leq \rangle$ be a quasi-ordered set and $x_0, \ldots, x_n \in L$. The sequence $x_0, \ldots, x_n$ is called a chain of length n from x to y if and only if $x = x_0 > x_1 > \cdots > x_n \simeq y$. An infinite sequence $x_0, x_1 \ldots$ is called an infinite ascending (or descending) chain from x_0 if and only if $x_0 < x_1 < \ldots$ (or $x_0 > x_1 > \ldots$).*

Definition 31 (Lattice homomorphism) *Let $\langle L, \wedge, \vee \rangle$ and $\langle K, \cap, \cup \rangle$ be lattices. A mapping $f : L \to K$ is a* lattice homomorphism *if f is join-preserving and meet-preserving, that is, for all x and y in L:*

1. *$f(x \vee y) = f(x) \cup f(y)$ and*
2. *$f(x \wedge y) = f(x) \cap f(y)$*

Definition 32 (Lattice isomorphism) *Let $\langle L, \wedge, \vee \rangle$ and $\langle K, \cap, \cup \rangle$ be lattices. A mapping $f : L \to K$ is a* lattice isomorphism *if f is a bijective lattice homomorphism.*

Proposition 6 *If $f : L \to K$ is a one-to-one homomorphism, then the sub-lattice $f(L)$ of K is isomorphic to L and we refer to f as an embedding of L into K.*

Proposition 7 *Let $\langle L, \wedge, \vee \rangle$ and $\langle K, \cap, \cup \rangle$ be lattices.*

- *a mapping $f : L \to K$ is order-preserving if it is a lattice homomorphism.*
- *a mapping $f : L \to K$ is order-isomorphism if and only if it is a lattice isomorphism.*

Note that if two lattices are isomorphic then for all practical purposes they are identical and differ only in the notation of their elements. In other words, an isomorphism faithfully mirrors the order structure.

2.2 ILP problem setting

The following definition, adapted from [NCdW97], defines the learning problem setting for ILP.

Definition 33 (ILP problem setting) *Given $\langle B, E \rangle$, where B is a set of clauses representing the background knowledge and E is the set of positive (E^+) and negative (E^-) examples such that $B \not\models E^+$, find a theory $\mathcal{H}$ such that $\mathcal{H}$ is complete and (weakly) consistent with respect to B and E. $\mathcal{H}$ is complete with respect to B and E^+ if $B \wedge \mathcal{H} \models E^+$. $\mathcal{H}$ is consistent with respect to B and E^- if $B \wedge \mathcal{H} \wedge E^- \not\models \Box$. $\mathcal{H}$ is weakly consistent with respect to B if $B \wedge \mathcal{H} \not\models \Box$.*

Note that in practice, due to the noise in the training examples, the completeness and consistency conditions are usually relaxed. For example, weak consistency is usually used and a noise threshold is considered which allows $\mathcal{H}$ to be inconsistent with respect to a certain proportion (or number) of negative examples. The following example is adapted from [LD93].

Example 1 *In Definition 33, let E^+, E^- and B be defined as follows:*

$$
\begin{aligned}
E^+ &= \{daughter(mary, ann), daughter(eve, tom)\} \\
E^- &= \{daughter(tom, ann), daughter(eve, ann)\} \\
B &= \{mother(ann, mary), mother(ann, tom), father(tom, eve), \\
&\quad father(tom, ian), female(ann), female(mary), \\
&\quad female(eve), male(pat), male(tom), \\
&\quad parent(X, Y) \leftarrow mother(X, Y), \\
&\quad parent(X, Y) \leftarrow father(X, Y)\}
\end{aligned}
$$

Then both theories $\mathcal{H}_1$ and $\mathcal{H}_2$ defined as follows:

$$\begin{aligned}
\mathcal{H}_1 &= \{daughter(X,Y) \leftarrow female(X), parent(Y,X)\} \\
\mathcal{H}_2 &= \{daughter(X,Y) \leftarrow female(X), mother(Y,X), \\
&\quad daughter(X,Y) \leftarrow female(X), father(Y,X)\}
\end{aligned}$$

are complete and consistent with respect to B and E. $\diamond$

2.3 Subsumption order

Plotkin [Plo71] introduced the subsumption order between clauses. This has been the theoretical basis for generalisation and specialisation in ILP. The general subsumption order on clauses, also known as θ-subsumption, is defined as follows.

Definition 34 (Subsumption on clauses) *Let C and D be clauses. We say C subsumes D, denoted by $C \succeq D$, if there exists a substitution θ such that $C\theta$ is a subset of D. C properly subsumes D, denoted by $C \succ D$, if $C \succeq D$ and $D \not\succeq C$. C and D are subsume-equivalent, denoted by $C \sim D$, if $C \succeq D$ and $D \succeq C$.*

The subsumption order on atoms, which is a special case of Definition 34, is defined as follows.

Definition 35 (Subsumption on atoms) *Let A and B be atoms. We say A subsumes B, denoted by $A \succeq B$, if there exists a substitution θ such that $A\theta = B$. A properly subsumes B, denoted by $A \succ B$, if $A \succeq B$ and $B \not\succeq A$. A and B are subsume-equivalent, denoted by $A \sim B$, if $A \succeq B$ and $B \succeq A$.*

Proposition 8 (Subsumption lattice for atoms) *Let $\mathcal{A}$ be the set of atoms in a language and $\succeq$ be the subsumption order as defined in Definition 35. Every finite subset of $\mathcal{A}$ has a* most general specialisation *(mgs), obtained from the unification algorithm, and a* least general generalisation *(lgg), obtained from the anti-unification algorithm. Thus $\langle \mathcal{A}, \succeq \rangle$ is a lattice.*

Proposition 9 (Subsumption lattice for clauses) *Let $\mathcal{C}$ be a clausal language and $\succeq$ be the subsumption order as defined in Definition 34. Then the equivalence classes* *of clauses in $\mathcal{C}$ and the $\succeq$ order define a lattice. Every pair of clauses C and D in the subsumption lattice have a least upper bound called* least general generalisation (lgg)*, denoted by $lgg(C, D)$ and a greatest lower bound called* most general specialisation (mgs)*, denoted by $mgs(C, D)$. Thus $\langle \mathcal{C}, \succeq \rangle$ is a lattice.*

2.4 Refinement operators

Shapiro [Sha83] introduced the framework of model inference and the concept of refinement operator as a function which computes a set of specialisations of a clause. He incorporated refinement operators in his model inference system MIS which was used for debugging definite logic programs. The concept of refinement operator has been the basis of many ILP systems. The following definition is a reminder of the concept of refinement operator and several properties of these operators.

Definition 36 (Refinement operator) *Let $\mathcal{C}$ be a clausal language and $\succeq$ be the subsumption order as defined in Definition 34. A (downward)* refinement operator *for $\langle \mathcal{C}, \succeq \rangle$ is a function ρ, such that $\rho(C) \subseteq \{D | C \succeq D\}$, for every $C \in \mathcal{C}$.*

- *The sets of* one-step refinements*,* n-step refinements *and* refinements *of some $C \in \mathcal{C}$ are respectively: $\rho^1(C) = \rho(C)$, $\rho^n(C) = \{D|$ there is an $E \in \rho^{n-1}(C)$ such that $D \in \rho(E)\}, n \geq 2$ and $\rho^*(C) = \rho^1(C) \cup \rho^2(C) \cup ..$*
- *A ρ-*chain *from C to D is a sequence $C = C_0, C_1, \ldots, C_n = D$, such that $C_i \in \rho(C_{i-1})$ for every $1 \leq i \leq n$.*
- *ρ is* locally finite *if for every $C \in \mathcal{C}$, $\rho(C)$ is finite and computable.*
- *ρ is* proper *if for every $C \in \mathcal{C}$, $\rho(C) \subseteq \{D|C \succ D\}$.*
- *ρ is* complete *if for every $C, D \in \mathcal{C}$ such that $C \succ D$, there is an $E \in \rho^*(C)$ such that $D \sim E$ (i.e. D and E are equivalent in the $\succeq$-order).*
- *ρ is* weakly complete *for $\langle \mathcal{C}, \succeq \rangle$ if $\rho^*(\square) = \mathcal{C}$, where $\square$ is the top element of $\mathcal{C}$.*
- *ρ is* non-redundant *if for every $C, D, E \in \mathcal{C}$, $E \in \rho^*(C)$ and $E \in \rho^*(D)$ implies $C \in \rho^*(D)$ or $D \in \rho^*(C)$.*

$$\begin{tikzcd}
C: & & p(x,y) \arrow[dl] \arrow[d] \arrow[dr] & & \\
\rho(C): & p(x,x) \arrow[d] & p(x,y) \leftarrow q(x,z) \arrow[dl] \arrow[d] \arrow[dr] & p(x,y) \leftarrow r(w,y) \arrow[dr] & \\
\rho^2(C): & p(x,x) \leftarrow q(x,z) \arrow[d] & \begin{array}{c} p(x,y) \leftarrow q(x,z), \\ q(z,w) \end{array} \arrow[d] & p(x,y) \leftarrow q(x,x) \arrow[d] & p(x,y) \leftarrow r(y,y) \arrow[d] \\
\rho^3(C): & p(x,x) \leftarrow q(x,y) \arrow[d] & \begin{array}{c} p(x,y) \leftarrow q(x,z), \\ q(z,x) \end{array} \arrow[d] & \begin{array}{c} p(x,y) \leftarrow q(x,x), \\ r(w,y) \end{array} \arrow[d] & \begin{array}{c} p(x,y) \leftarrow r(y,y), \\ q(x,z) \end{array} \arrow[d] \\
\rho^n(C): & \ldots & \ldots & \ldots & \ldots
\end{tikzcd}$$

$$\rho^*(C) = \rho^0(C) \cup \rho^1(C) \cup \rho^2(C) \ldots$$

Figure 2.1: Part of a refinement graph representing (downward) refinements of a clause.

- *ρ is* ideal *if it is locally finite, proper and complete.*
- *ρ is* optimal *if it is locally finite, non-redundant and weakly complete.*

We can define analogous concepts for the dual case of an upward refinement operator.

Example 2 *Figure 2.1 shows part of a (downward) refinement graph for the subsumption order. In this graph clause $p(x,y)$ is refined either by unifying variables or by adding literals. The refinement operator presented by this graph is not complete as it does not include all possible refinements. It is proper as the graph does not contain cycles. It is redundant because it does not have a tree structure and there is more than one path from $p(x,y)$ to $p(x,x) \leftarrow q(x,z)$.* $\diamond$

The following definition for binary refinement is adapted [3] from [MM99].

Definition 37 (Binary refinement operator) *Let $\langle G, \succeq \rangle$ be a quasi-ordered set. A (downward)* binary refinement operator *for $\langle G, \succeq \rangle$ is a function $\rho : G^2 \rightarrow 2^G$, such that $\rho(C,D) \subseteq \{E | C \succeq E, D \succeq E\}$, for every $C \in G$.*

[3] Note that in [MM99], $\rho^n(C,D)$ is defined as the binary refinement of a pair of clauses F and H which can be from any previous steps of binary refinement of C and D. However, in our definition F and H can only be from the previous step, i.e. $\rho^{n-1}(C,D)$. As discussed in Chapter 6, this is used to define a Markov chain of refinements where the next state depends only on the current state.

- *The sets of* one-step refinements*,* n-step refinements *and* refinements *of some $C, D \in G$ are respectively: $\rho^1(C,D) = \rho(C,D)$, $\rho^n(C,D) = \{E|$ there is an $F \in \rho^{n-1}(C,D)$ and an $H \in \rho^{n-1}(C,D)$ such that $E \in \rho(F,H)\}, n \geq 2$ and $\rho^*(C,D) = \rho^1(C,D) \cup \rho^2(C,D) \cup ..$*

- *A ρ-*chain *(C,D) to E is a sequence $(C,D) = (C_0,D_0),(C_1,D_1),\dots,(C_m,D_m)$, such that $E = C_m$ or $E = D_m$ and $C_i, D_i \in \rho(C_{i-1},D_{i-1})$ for every $1 \leq i \leq m$.*

- *ρ is* locally finite *if for every $C, D \in G$, $\rho(C,D)$ is finite and computable.*

- *ρ is* proper *if for every $C, D \in G$, $\rho(C,D) \subseteq \{E|C \succ E, D \succ E\}$.*

- *ρ is* complete *if for every $B, C, D \in G$ such that $C \succ B$, $D \succ B$ there is an $E \in \rho^*(C,D)$ such that $B \sim E$ (i.e. B and E are equivalent in the $\succeq$-order).*

- *ρ is* weakly complete *for $\langle G, \succeq \rangle$ if $\rho^*(\Box,\Box) = G$, where $\Box$ is the top element of G.*

- *ρ is* non-redundant *if for every $B, C, D, E, F \in G$, $F \in \rho^*(B,C)$ and $F \in \rho^*(D,E)$ implies $B, C \in \rho^*(D,E)$ or $D, E \in \rho^*(B,C)$.*

- *ρ is* ideal *if it is locally finite, proper and complete.*

- *ρ is* optimal *if it is locally finite, non-redundant and weakly complete.*

We can define analogous concepts for the dual case of an upward binary refinement operator.

2.5 Golem and RLGG

Plotkin [Plo71] investigated the problem of finding the least general generalisation (*lgg*) for clauses ordered by subsumption. The notion of *lgg* is important for ILP since it forms the basis of generalisation algorithms which perform a bottom-up search of the subsumption lattice. Plotkin also defined the notion of relative least general generalisation of clauses (*rlgg*) which is the *lgg* of the clauses relative to clausal background knowledge B.

Definition 38 (Relative subsumption [NCdW97]) *Let C and D be clauses and B be a set of clauses. We say C subsumes D relative to B, denoted by $C \succeq_B D$, if there exists a substitution θ such that $B \models \forall(C\theta \rightarrow D)$. The $\succeq_B$ order is called relative subsumption and B is the background knowledge of this order. Least general generalisation with respect to relative subsumption is called relative least general generalisation (rlgg).*

Note that in Definition 38, $\rightarrow$ is used to denote the implication connective and $\forall(C\theta \rightarrow D)$ will usually not be a clause.

Proposition 10 (Existence of rlgg in $\mathcal{C}$ [NCdW97]) *Let $\mathcal{C}$ be a clausal language and $B \subseteq \mathcal{C}$ be a finite set of ground literals. Then every non-empty set $S \subseteq \mathcal{C}$ of clauses has an rlgg in $\mathcal{C}$.*

The cardinality of the *lgg* of two clauses is bounded by the product of the cardinalities of the two clauses. However, the *rlgg* is potentially infinite for arbitrary B. When B consists of ground unit clauses only the *rlgg* of two clauses is finite. However the cardinality of the *rlgg* of m clauses relative to n ground unit clauses has worst-case cardinality of order $O(n^m)$, making the construction of such *rlgg*'s intractable.

The ILP system Golem [MF90] is based on Plotkin's notion of relative least general generalisation of clauses (*rlgg*). Golem uses extensional background knowledge to avoid the problem of non-finite *rlggs*. Extensional background knowledge B is generated from intensional background knowledge B' by generating all ground unit clauses derivable from B' in at most h resolution steps. The parameter h is provided by the user. The *rlggs* constructed by Golem were forced to have only a tractable number of literals by requiring ij-determinacy.

The ij-determinacy is equivalent to requiring that predicates in the background knowledge must represent functions. j-determinate clauses are constrained to having at most j variables in any literal. ij-determinate clauses are further restricted that each variable has depth at most depth i. For variable v the depth $d(v)$ is defined recursively as follows.

Definition 39 (Depth of variables [Mug95]) *Let C be a definite clause and v be* *a variable in C. Depth of v is defined as follows:*

$$d(v) = \begin{cases} 0 & \text{if } v \text{ is in the head of } C \\ (max_{u \in U_v} d(u)) + 1 & \text{otherwise} \end{cases}$$

where U_v are the variables in atoms in the body of C containing v.

Even with the determinacy constraint, *rlgg*'s can be very long clauses containing irrelevant literals. Golem employs a negative-based reduction algorithm in which negative examples are used to reduce the size of the hypothesised clauses. The negative-based reduction algorithm can be summarised as follows. Given a clause $a \leftarrow b_1, \ldots, b_n$, find the first literal, b_i such that the clause $a \leftarrow b_1, \ldots, b_i$ covers no negative examples. Prune all literals after b_i and move b_i and all its supporting literals to the front, yielding a clause $a \leftarrow S_i, b_i, T_i$, where S_i is a set of supporting literals needed to introduce the input variables of b_i and T_i is $b_1, \ldots, b_{i-1}$ with S_i removed. Then reduce this new clause in the same manner and iterate until the clause length remains the same within a cycle.

The following example shows how *rlgg*'s are used in Golem. This example is adapted from [LD93].

Example 3 *Let E^+, E^- and B be defined as in Example 1 and e_1, e_2 be two positive examples and K be the conjunction of ground facts from B defined as follows:*

$$\begin{aligned} e_1 &= daughter(mary, ann) \\ e_2 &= daughter(eve, tom) \\ K &= parent(ann, mary), parent(ann, tom), parent(tom, eve), \\ &\quad parent(tom, ian), female(ann), female(mary), female(eve) \end{aligned}$$

Then we have

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$$

which generates the following clause:

$$\begin{aligned} daughter(V_{m,e}, V_{a,t}) \quad \leftarrow \quad & parent(ann, mary), parent(ann, tom), parent(tom, eve), \\ & parent(tom, ian), female(ann), female(mary), female(eve), \\ & parent(ann, V_{m,t}), parent(V_{a,t}, V_{m,e}), parent(V_{a,t}, V_{m,i}), \\ & parent(V_{a,t}, V_{t,e}), parent(V_{a,t}, V_{t,i}), parent(tom, V_{e,i}), \\ & female(V_{a,m}), female(V_{a,e}), female(V_{m,e}) \end{aligned}$$

After reducing this clause we have the following clause:

$$daughter(X, Y) \quad \leftarrow \quad female(X), parent(Y, X)$$

Golem was the first ILP system which was successfully applied to real-world problems and led to scientific discoveries [MKS92]. However, the determinacy condition in Golem is not met in many real-world applications, including the learning of chemical properties from atom and bond descriptions.

2.6 Progol and Mode-Directed Inverse Entailment (MDIE)

One of the motivations of the ILP system Progol [Mug95] was to overcome the determinacy limitation of Golem. Progol extends the idea of inverting resolution proofs used in the systems Duce [Mug87] and Cigol [MB88] and uses the general case of Inverse Entailment which is based on the model-theory which underlies proof. Mode-Directed Inverse Entailment (MDIE) is the basis of the ILP system Progol. The MDIE setting can be summarised as follows. The input to an MDIE system is $\langle M, B, E \rangle$ where M is a set of mode statements, B is a logic program representing the background knowledge and E is set of examples. M can be viewed as a set of metalogical statements used to define the hypothesis language $\mathcal{L}_M$. The aim of the system is to find a set of consistent hypothesised clauses $\mathcal{H}$ such that for each clause $H \in \mathcal{H}$ there is at least one positive example $e \in E$ such that the following holds.

$$B, H \models e$$

For any B, H, e this is equivalent to the following.

$$B, \neg e \models \neg H$$

This form allows hypotheses to be derived from B and e using standard Prolog theorem proving techniques. Since $\neg H$ takes the form of a ground conjunction of literals, for any finitely bound hypothesis language $\mathcal{L}_M$ there is a maximal ground conjunction $\bot_e$ for which the following holds.

$$B, \neg e \models \neg \bot_e \models \neg H$$

Having selected an example e and constructed $\bot_e$ Progol conducts a refinement graph search which considers hypotheses H in the interval

$$\Box \succ H \succeq \bot_e$$

where "$\succeq$" denotes θ-subsumption and $\Box$ is the empty clause.

Progol uses MDIE to develop a most specific clause $\bot$ for each positive example, within the user-defined mode language, and uses this to guide an A^*-like search through clauses which subsume $\bot$.

It was known (Example 30 in [Mug95]) that Progol's refinement operator is not complete due to the choice of ordering of $\bot_e$. A second type of incompleteness, which is due to the fact that each literal from $\bot$ is selected only once, was shown by [BS99]. The authors of [BS99] also defined a subsumption order, called weak subsumption, for characterising Progol's refinement space. However, weak subsumption cannot capture the ordering aspect of Progol's refinement. In this thesis, we characterise the structure of the search space and clause refinement in Progol. For this purpose, we introduce the concept of bounded subsumption, i.e. subsumption relative to a bottom clause and we show that this can fully capture Progol's refinement. Progol's refinement is discussed with more details and examples in the next chapter.

2.7 Summary

The purpose of this chapter was to provide the key concepts from logic-based machine learning and Inductive Logic Programming (ILP) which are are needed in different parts of this thesis. We gave an introduction to ILP in the context of Machine Learning and described the ILP problem setting and important elements such as subsumption and refinement operators. We also briefly discussed the ILP systems Golem and Progol which are important in the context of this thesis.

Chapter 3

Hypothesis space and bounded subsumption

In this chapter we first give an overview of bounded subsumption and related results. In Section 3.2 we review clause refinement in Progol as a representative of ILP systems which use a bottom clause to constrain the search. We give examples of Progol's incompleteness with respect to the general subsumption order. Ordered clauses and sequential subsumption are discussed in Section 3.3. In order to characterise refinement in a Progol-like ILP system, Section 3.4 defines subsumption order relative to a bottom clause and describes the properties of this subsumption order. Related work is discussed in Section 3.5. Section 3.6 summarises the chapter.

3.1 Bounded subsumption, an overview

The purpose of this section is to give an overview of bounded subsumption and related results in order to motivate the theoretical developments in this thesis. As shown in Section 2.2, an ILP problem can be characterised as a search for a complete and (weakly) consistent theory through the space of clauses. This search space is ordered by the generality order between clauses (or set of clauses) and the most natural generality order to consider for this purpose is entailment. However, in practice we need to consider alternative generality orders as entailment is known to be undecidable. For this reason, instead of entailment, θ-subsumption has been used in the early theoretical developments (e.g. [Plo71]) as well as system developments (e.g. MIS [Sha83]).

Unlike entailment, θ-subsumption is decidable although it is shown to be NP-complete [GJ79]. Table 3.1 compares Entailment, θ-subsumption and several other generality orders across a number of dimensions. As shown in this table, another advantage of θ-subsumption over entailment is the existence of lgg and a lattice structure which can facilitate the search. The subsumption graph theory has been viewed as the main theoretical foundation of ILP and searching the subsumption lattice, either by using lgg (e.g. Golem [MF90]) or refinement operators (e.g. MIS [Sha83] and Progol [Mug95]) has been the basis of many ILP systems.

The advantages of θ-subsumption mentioned above are at the cost of incompleteness with respect to entailment. This incompleteness is demonstrated in the following example.

Example 4 (Incompleteness type 0) *Let C, D and E be clauses as defined below.*

$$\begin{aligned} C &= nat(s(X_1)) \leftarrow nat(X_1). \\ D &= nat(s(s(X_2))) \leftarrow nat(s(X_2)). \\ E &= nat(s(s(X_3))) \leftarrow nat(X_3). \end{aligned}$$

Every model for C is also a model for D and E and therefore we have $C \models D$ and $C \models E$. We also have $C \succeq D$, however, there exist no substitution θ such that $C\theta \subseteq E$ and therefore $C \not\succeq E$. ◇

Example 4 shows the incompleteness of θ-subsumption with respect to entailment (incompleteness type 0). Nevertheless, θ-subsumption has been widely used in ILP systems despite this incompleteness which only occurs when learning recursive clauses similar to the example above. In other words, this incompleteness was not a limitation in most applications and θ-subsumption has been regarded as the main generality order in ILP. However, θ-subsumption is NP-complete and one could argue that by further restricting θ-subsumption it might be possible to obtain a generality order which is more efficient and still have an acceptable degree of completeness for real-world applications. This is similar to the trade-off between efficiency and completeness which already exists for θ-subsumption with respect to entailment. We show that

Generality order	**Condition**	**Existence of lgg & lattice structure**	**Existence of ideal refinement operators**	**Time complexity**	**Incompleteness (with respect to entailment)**
Entailment ($\models$) on Horn clauses ([NCdW97])	$C \models D$ if every model of C is a model of D	No	No	Undecidable	No
θ-subsumption ($\succeq_\theta$) on Horn clauses ([NCdW97])	$C \succeq_\theta D$ if $C\theta \subseteq D$ for some substitution θ	Yes, but the length grows very rapidly	No	NP-Complete	Type 0
Weak Subsumption ($\succeq_w$) on Horn clauses ([BS99])	$C \succeq_w D$ if $C\theta \subseteq D$ for some substitution θ that does not unify literals and $\theta_\perp(D)\theta = \theta_\perp(C)$	No	Yes	NP-Complete	Type 0, Type 2
Subsumption under Object Identity ($\succeq_{OI}$) on Horn clauses ([ELMS96, AR99])	$C \succeq_{OI} D$ if $C\theta \subseteq D$ for some injective substitution θ	No	Yes	NP-Complete	Type 0, Type 2, Type 3
Ordered Subsumption ($\succeq_o$) on ordered Horn clauses ([KOHH06])	$\overrightarrow{C} \succeq_o \overrightarrow{D}$ if $\overrightarrow{C}\theta$ is a monotonic subsequence of $\overrightarrow{D}$ for some substitution θ	No	No	NP-Complete	Type 0, Type 1
s-subsumption ($\sqsubseteq_s$) on simple sequences ([LD04, Lee06])	$p \sqsubseteq_s q$ if $p\theta$ is a simple subsequence of q for some substitution θ	No	Optimal refinement operators are defined	Polynomial	Type 0, Type 1, Type 2
Sequential Subsumption ($\succeq_s$) on ordered Horn clauses (Section 3.3)	$\overrightarrow{C} \succeq_s \overrightarrow{D}$ if $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$ for some substitution θ	No	Yes	Polynomial	Type 0, Type 1, Type 2
Bounded Subsumption ($\succeq_\perp$) on ordered Horn clauses (Section 3.4)	$\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ if $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$ for some substitution θ relative to $\perp$	Yes and the length is bounded by $\|\perp\|$	Yes	Linear	Type 0, Type 1, Type 2
Injective Subsumption ($\succeq^i$) on ordered Horn clauses (Section 5.4)	$\overrightarrow{C} \succeq^i \overrightarrow{D}$ if $\overrightarrow{C}\theta$ is an injective subset of $\overrightarrow{D}$ for some substitution θ	No	Yes	Polynomial	Type 0, Type 2
Injective Bounded Subsumption ($\succeq^i_\perp$) on ordered Horn clauses (Section 5.4)	$\overrightarrow{C} \succeq^i_\perp \overrightarrow{D}$ if $\overrightarrow{C}\theta$ is an injective subset of $\overrightarrow{D}$ for some substitution θ relative to $\perp$	Yes and the length is bounded by $\|\perp\|$	Yes	Linear	Type 0, Type 2

Table 3.1: A comparison of several generality orders for clauses.

the subsumption orders relative to a bottom clause, or simply bounded subsumption orders, introduced in this thesis have a linear time complexity and at the same time have an acceptable degree of completeness such that they have been successfully used in many real-world applications. Bounded subsumption orders also have other interesting properties, e.g. efficient lgg operators and the existence of ideal refinement operators which we briefly discuss in this section. But first we look back at another limitation of θ-subsumption and show how this has been addressed by alternative subsumption orders.

Apart from time complexity, another problem of θ-subsumption is that a long clause can θ-subsume a short clause while it is usually expected that a long clause is more specific. This is shown in the following example.

Example 5 *Let clause C be $C = p(X) \leftarrow q(X,X)$, then the following clauses all subsume C but they are longer than C.*

$$\begin{aligned}
C'_1 &= p(X') \leftarrow q(X',Y'), q(Y',X') \\
C'_2 &= p(X') \leftarrow q(X',Y'), q(Y',Z'), q(Z',X') \\
C'_3 &= p(X') \leftarrow q(X',Y'), q(Y',Z'), q(Z',W'), q(W',X')
\end{aligned}$$

This example is also related to infinite ascending chains and shows that clause C has no finite complete set of downward covers under θ-subsumption and therefore ideal refinement operators do not exist for θ-subsumption. As shown in this example, a long clause can θ-subsume a short clause because there exist a substitution θ which can unify several literals from the long clause into the same literal from the short clause.

However, there are alternatives to θ-subsumption which have solved this problem by putting some constraints on the substitution. Weak subsumption [BS99] does not allow substitutions that unify literals. For example, the clause $\mathrm{p}(X') \leftarrow \mathrm{q}(X',Y'), \mathrm{q}(Y',X')$ θ-subsumes $C = \mathrm{p}(X) \leftarrow \mathrm{q}(X,X)$, but it does not weakly subsume it because substitution $\{X'/X, Y'/X\}$ which unifies literals $\mathrm{q}(X',Y')$ and $\mathrm{q}(Y',X')$ is not allowed in weak subsumption. Weak subsumption, therefore, introduces an injective mapping between the literals. Similarly, subsumption under object identity (e.g. [ELMS96, AR99]) requires the injective mapping between all variables (objects). The subsumption under object identity, therefore, implies weak subsumption, as the injection at object level entails the injection at literal level. Injective mapping between literals resolves the problem of infinite covers and it has been shown that ideal refinement operators exist for weak subsumption ([BS99]) as well as for subsumption under object identity ([ELMS96]). Because variables cannot be unified, the subsumption under object identity introduces a new type of incompleteness. We refer to this as incompleteness type 3 which is demonstrated in the following example [1].

Example 6 (Incompleteness type 3) *Let C, D and E be clauses as defined below.*

$$\begin{aligned} C &= p(X,X) \leftarrow q(X,T), r(T,X). \\ D &= p(X,X) \leftarrow q(X,X), r(X,X), s(Y,a). \\ E &= p(X,X) \leftarrow q(X,X), r(X,X), s(Y,a), t(V). \end{aligned}$$

Clause D subsumes clause E with regards to both θ-subsumption and OI-subsumption, i.e. $D \succeq_\theta E$ and $D \succeq_{OI} E$. We also have $C \succeq_\theta D$, however, $C \not\succeq_{OI} D$ because this requires unification of variables X and T which is not allowed under OI-subsumption. $\diamond$

The main theoretical part of this thesis is concerned with the bounded subsumption order introduced in this thesis. The initial motivation for the introduction of bounded subsumption order was to characterise the refinement space of ILP systems such as Progol and Aleph which use a bottom clause to restrict the search space. However, this analysis is not limited to these systems and the framework of bounded subsumption is interesting in its own right and can be adapted by new systems. For example, efficient operators and algorithms exist for bounded subsumption which have been used in GA-Progol and ProGolem (see Chapter 7).

Bounded subsumption is a restricted form of θ-subsumption where the choice and order of literal mappings are decided with respect to a bottom clause. In order to define subsumption relative to a bottom clause we need to adapt concepts such as ordered clauses

[1] This example is adapted from Example 5.24 in [De 08].

and subsequences. An ordered clause denoted by $\overrightarrow{C}$ is a sequence of literals where the order and duplication of literals matter. In bounded subsumption all clauses are regarded as ordered clauses and are (sequential) generalisations of the bottom clause, i.e. all clauses are in language $\overrightarrow{\mathcal{L}}_\perp$ (see Definition 53). Moreover, when mapping literals between two clauses, only those pairs of literals can be considered which correspond to the same literal of the bottom clause. For example, let the bottom clause be $\overrightarrow{\perp} = p(X) \leftarrow q(X), r(X), s(X, Y), s(Y, X)$ and $\overrightarrow{C} = p(V_1) \leftarrow r(V_2), s(V_6, V_7)$, $\overrightarrow{D} = p(V_1) \leftarrow r(V_1), s(V_6, V_1)$ and $\overrightarrow{E} = p(V_1) \leftarrow r(V_1), s(V_4, V_5)$ be ordered clauses as shown in Figure 3.1. This figure shows mapping of literals with respect to a bottom clause. Literals from $\overrightarrow{C}$ can be mapped to the literals from $\overrightarrow{D}$ respectively as they correspond to the same literals from $\overrightarrow{\perp}$. The third literal from $\overrightarrow{C}$ and the third literal from $\overrightarrow{E}$ cannot be mapped together as they correspond to different literals from $\overrightarrow{\perp}$. This mapping is decided using substitution relative to $\perp$ (see Definition 54), which in this example is $\theta_\perp = \{V_2/V_1, V_3/V_1, V_4/V_1, V_7/V_1, V_3/V_2, V_4/V_2, V_7/V_2, V_7/V_3, V_7/V_4, V_6/V_5\}$. $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\perp$ ($\overrightarrow{C} \succeq_\perp \overrightarrow{D}$) since there is a substitution $\theta = \{V_2/V_1, V_7/V_1\} \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. However, $\overrightarrow{C}$ does not subsume $\overrightarrow{E}$ relative to $\perp$ since there is no substitution $\theta \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{E}$. Note that $\overrightarrow{C}$ subsumes $\overrightarrow{E}$ with respect to θ-subsumption, i.e. $C \succeq_\theta E$, however, $\overrightarrow{C} \not\succeq_\perp \overrightarrow{E}$.

This example shows that the set of bounded subsumption relations is a subset of θ-subsumption relations and therefore bounded subsumption is incomplete with respect to θ-subsumption. On the other hand, the bounded subsumption framework is computationally more efficient that θ-subsumption. For example, in this thesis we show that the lattice of bounded subsumption is isomorphic to an atomic lattice and therefore bounded subsumption testing can be reduced to atomic subsumption which is decidable in linear time whereas θ-subsumption testing is known to be NP-complete [GJ79].

The morphism between the lattice of bounded subsumption and an atomic lattice is discussed in Section 4.2. Figure 3.2 shows how bounded subsumption can be mapped to atomic subsumption. Moreover, we show that this atomic lattice is isomorphic to the lattice of partitions and the atoms can be encoded as partitions (Section 4.3). This encoding is the basis of several refinement operator and algorithms (Chapters 5 and 7). We also show (Section 5.2) that ideal refinement operators exist for bounded sub-

$$\begin{tikzcd}
\overrightarrow{C} = & p(V_1) \leftarrow \arrow[d] & r(V_2), \arrow[d] \arrow[dd, r, dotted] & s(V_6, V_7). \arrow[d] \arrow[dd, rr, dotted] & & \\
\overrightarrow{D} = & p(V_1) \leftarrow \arrow[d, dotted] & r(V_1), \arrow[d, r, dotted] & s(V_6, V_1). \arrow[d, rr, dotted] & & \\
\overrightarrow{\bot} = & p(X) \leftarrow & q(X), & r(X), & s(X,Y), & s(Y,X).
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\overrightarrow{C} = & p(V_1) \leftarrow \arrow[d] & r(V_2), \arrow[d] \arrow[dd, r, dotted] & s(V_6, V_7). \arrow[d, dashed] \arrow[dd, rr, dotted] & & \\
\overrightarrow{E} = & p(V_1) \leftarrow \arrow[d, dotted] & r(V_1), \arrow[d, r, dotted] & s(V_4, V_5). \arrow[d, r, dotted] & & \\
\overrightarrow{\bot} = & p(X) \leftarrow & q(X), & r(X), & s(X,Y), & s(Y,X).
\end{tikzcd}$$

(b)

Figure 3.1: Subsumption relative to $\bot$ (a) $\vec{C}$ subsumes $\vec{D}$ relative to $\bot$ ($\vec{C} \succeq_\bot \vec{D}$) since there is a substitution $\theta = \{V_2/V_1, V_7/V_1\} \subseteq \theta_\bot$ such that $\vec{C}\theta$ is a subsequence of $\vec{D}$ (b) $\vec{C}$ does not subsume $\vec{E}$ relative to $\bot$ since there is no substitution $\theta \subseteq \theta_\bot$ such that $\vec{C}\theta$ is a subsequence of $\vec{E}$. Note that $\vec{C} \succeq_\theta \vec{E}$ but $\vec{C} \not\succeq_\bot \vec{E}$.

sumption and that, by contrast with general subsumption, efficient least and minimal generalisation operators can be designed for bounded subsumption.

Hence, the main advantages of bounded subsumption over θ-subsumption can be summarised as follows:

1. morphism with the atomic lattice and that bounded subsumption testing is decidable in linear time (Sections 4.2),
2. existence of ideal refinement operators (Section 5.2) and
3. efficient least and minimal generalisation operators (Sections 5.3 and 7.3.1).

However, the practical disadvantages of bounded subsumption over θ-subsumption are less obvious. There are only two known cases of the incompleteness of bounded subsumption with respect to θ-subsumption:

Incompleteness type 1 due to the choice of ordering in the bottom clause and the variable dependencies in the literals and

Incompleteness type 2 due to the fact that each literal from $\bot$ can be selected only once.

$$\begin{tikzcd}
A_1 = & \vee(p(V_1), \arrow[d] & \_, & \neg r(V_3), \arrow[d] & \_, & \neg s(V_6, V_7)) \arrow[d] \\
A_2 = & \vee(p(V_1), \arrow[d, dotted] & \_, & \neg r(V_1), \arrow[d, dotted] & \_, & \neg s(V_6, V_1)) \arrow[d, dotted] \\
a(\overrightarrow{\bot}) = & \vee(p(X), & \neg q(X), & \neg r(X), & \neg s(X, Y), & \neg s(Y, X))
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
A_1 = & \vee(p(V_1), \arrow[d] & \_, & \neg r(V_3), \arrow[d] & \_, & \neg s(V_6, V_7)) \arrow[d, dotted] \\
A_3 = & \vee(p(V_1), \arrow[d, dotted] & \_, & \neg r(V_1), \arrow[d, dotted] & \neg s(V_4, V_5), \arrow[d, dotted] & \_) \arrow[d, dotted] \\
a(\overrightarrow{\bot}) = & \vee(p(X), & \neg q(X), & \neg r(X), & \neg s(X, Y), & \neg s(Y, X))
\end{tikzcd}$$

(b)

Figure 3.2: Subsumption relative to a bottom clause can be mapped to atomic subsumption. Atoms A_1, A_2 and A_3 correspond to ordered clauses $\vec{C}$, $\vec{D}$ and $\vec{E}$ in Figure 3.1. (a) $\vec{C} \succeq_\bot \vec{D}$ and $A_1 \succeq A_2$ (b) $\vec{C} \not\succeq_\bot \vec{E}$ and $A_1 \not\succeq A_3$.

Incompleteness type 1 is demonstrated in the following example [2].

Example 7 (Incompleteness type 1) *Let B contain definitions for decrementation (dec), addition (plus) and the clause $mult(0, X, 0) \leftarrow$ with appropriate mode declarations M and let the example e be the clause $mult(1, 1, 1) \leftarrow$. Then $\bot$ is the clause*

$$\begin{aligned} mult(A, A, A) \quad \leftarrow \quad & dec(A, B), plus(B, A, A), plus(B, B, B), \\ & mult(B, A, B), mult(B, B, B). \end{aligned}$$

Now consider clause $\overrightarrow{C}$:

$$\overrightarrow{C} = mult(U, V, W) \leftarrow dec(U, X), mult(X, V, Y), plus(Y, V, W).$$

Clause C θ-subsumes $\bot$, but given the ordering over $\bot$ there will be no substitution θ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{\bot}$. Hence, $\overrightarrow{C}$ is not in the bounded subsumption lattice, e.g. given this bottom clause there will be no element of Progol's refinement space containing this clause or a subsume-equivalent of this clause. $\diamond$

Note that in Example 7, clause $\overrightarrow{C}$ will appear in the lattices bounded by the bottom clauses generated from other examples (e.g. $e = mult(3, 5, 15) \leftarrow$) and therefore the search can eventually find clause $\overrightarrow{C}$.

[2] This example is a revised version of Example 30 in [Mug95].

In this thesis we also introduce (Section 5.4) injective subsumption relative to $\perp$ (or simply injective bounded subsumption) which is based on injective subset rather than subsequence. We show that the incompleteness type 1 can be addressed by a refinement operator which is based on injective bounded subsumption (see Example 34).

Example 8 (Incompleteness type 2) *Let B contain definitions for append and partition (part) with appropriate mode declarations M for learning quick sort (qs) and let the example e be the clause $qs([1],[1]) \leftarrow$. Then $\perp$ is the clause*

$$qs([U|V],W) \leftarrow par(V,X,X), qs(X,Y), append(Y,[U|Y],W).$$

Now consider target clause $\overrightarrow{C}$:

$$\overrightarrow{C} = qs([A|B],G) \quad \leftarrow \quad par(B,A,C,D), qs(C,E), qs(D,F), append(E,[A|F],G).$$

Clause C θ-subsumes $\perp$, but given that each literal of $\perp$ can be selected only once, $\overrightarrow{C}$ is not in the bounded subsumption lattice, e.g. given this bottom clause, there will be no element of Progol's refinement space containing this clause or a subsume-equivalent of this clause. $\diamond$

To our knowledge Example 8 represent the only known case where the incompleteness of bounded subsumption with respect to θ-subsumption could have any practical implication. Note that in Example 8, the target clause $\overrightarrow{C}$ will appear in the lattices bounded by the bottom clauses generated from examples with lists of length of more than one (e.g. $e = qs([2,1],[1,2]) \leftarrow$) and therefore the search can eventually find clause $\overrightarrow{C}$. Moreover, this incompleteness can be compared to the incompleteness of θ-subsumption with respect to entailment (Example 4) which has been widely accepted due to the computability. It has also been argued [BS99] that incompleteness type 2 is not a drawback as it can be justified by the examples and the Minimum Description Length (MDL) heuristic.

The bounded subsumption order described above works on ordered clauses. Similarly, ordered subsumption [KOHH06] is defined for ordered clauses and is based on the concept of subsequence. However, the subsequence relation considered in [KOHH06], assumes a mapping function which is monotonically increasing (rather than strictly increasing mapping function considered in our definition of subsequence). This means that in ordered subsumption, several literals from the first clause can be mapped to the same literal from the second clause. However, as already discussed in this section, in the bounded subsumption distinct literals from the first clause can not be mapped to the same literal from the second clause. Hence, ordered subsumption does not have the incompleteness type 2 described above. However, as shown in [KOHH06], there exist no least generalisation under ordered subsumption and also the time complexity is NP-complete.

Another subsumption order which is closely related to bounded subsumption is s-subsumption defined for simple sequences in SeqLog [LD04, Lee06]. SeqLog is a logical language for representing and mining sequential patterns in databases. A simple sequence is defined as a sequence of atoms whereas a complex sequence is a sequence of atoms separated by (direct or indirect) 'successor' operators (see Section 3.5). It is shown [Lee06] that time complexity of s-subsumption is polynomial but the general SeqLog subsumption among complex sequences is NP-complete. It is also shown that lgg does not exist for any pair of simple sequences or complex sequences and subsumption for simple or complex sequences do not form lattices. However, optimal refinement operators are defined for simple and complex sequences. The sequential subsumption on ordered clauses defined in this thesis (Section 3.3) is similar to s-subsumption on simple sequences.

According to Table 3.1, lgg only exist for θ-subsumption and bounded subsumption and therefore in this table only these generality orders form a lattice. The cardinality of the lgg of two clauses under θ-subsumption is bounded by the product of the cardinalities of the two clauses. For bounded subsumption, the length of lgg is bounded by the cardinality of bottom clause.

In this table weak subsumption and bounded subsumption are defined with respect to a bottom clause. Like bounded subsumption, weak subsumption was initially introduced to characterise clause refinement in Progol. However, weak subsumption only characterises incompleteness type 2 and it does not capture the incompleteness due to the ordering of the literals. In weak subsumption, clauses are assumed to be pairs of $\langle C, \theta_\perp(C) \rangle$ such that $C\theta_\perp(C) \subseteq \perp$. Note that for a given clause C there might be several distinct substitutions θ_i such that $C\theta_i \subseteq \bot$ and therefore distinct hypothesis should be considered as pairs of $\langle C, \theta_i \rangle$. Similarly, the selection function and substitution θ which make $\overrightarrow{C}\theta$ a subsequence of another clause are not unique. Hence, the mappings for sequential subsumption and ordered subsumption are not unique, lgg does not exist and these generality orders do not form a lattice. On the other hand, in bounded subsumption, clauses in $\overrightarrow{\mathcal{L}}_\bot$ are uniquely defined using a substitution relative to $\bot$ ($\theta \subseteq \theta_\bot$) and variables in each clause $\overrightarrow{C} \in \overrightarrow{\mathcal{L}}_\bot$ implicitly define a unique mapping (see Definition 53). As shown in Chapter 4, each pair of ordered clauses have a most general specialisation and a least general generalisation under bounded subsumption and the bounded subsumption forms a lattice.

As shown in Table 3.1, only bounded subsumption and injective bounded subsumption have linear time complexities and among these, injective bounded subsumption is more complete. However, injective bounded subsumption is also more redundant than bounded subsumption (Section 5.4). In Chapter 7 we describe implementation of refinement operators based on both bounded subsumption and injective bounded subsumption. More details about sequential and bounded subsumption orders and relevant refinement operators are discussed throughout this thesis.

3.2 Clause refinement in Progol

In this section we study clause refinement in ILP systems in which the hypothesis space is bounded by a bottom clause. In particular we discuss refinement in Progol as a representative of these ILP systems.

The learning algorithm in Progol [Mug95] works by successive construction of definite clause hypotheses $\mathcal{H}$ from a language $\mathcal{L}$. $\mathcal{H}$ must explain the examples E in terms of background knowledge B. Each clause H in $\mathcal{H}$ is generated as follows. First an uncovered positive example e is selected and a most-specific clause or bottom clause $\bot$ associated with e is constructed. Then a search is performed through the graph defined by the refinement ordering $\succeq$ bounded below by the bottom clause $\bot$. Progol uses mode declarations to constrain the search for clauses which subsume $\bot$. The following are definitions for mode declaration (M), definite mode language ($\mathcal{L}(M)$)

and depth-bounded mode language ($\mathcal{L}_i(M)$) as described in [Mug95].

Definition 40 (Mode declaration M) *A mode declaration has either the form modeh (n,atom) or modeb(n,atom) where n, the recall, is either an integer, $n > 1$, or '*' and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type, where type is a constant. If m is a mode declaration then $a(m)$ denotes the atom of m with place-markers replaced by distinct variables. The sign of m is positive if m is a modeh and negative if m is a modeb.*

In Definition 40, the recall is used to bound the number of alternative solutions for instantiating the atom. A recall of '*' indicates all solutions [3].

Example 9 *The following are examples of mode declarations.*

modeh(,reverse(+list,-list))*	*modeb(*,+any= #any)*
modeb(,append(-list,+list,+list)*	*modeb(1,append(+list,[+any],-list))*
modeh(1,plus(+int,+int,-int))	*modeb(4,(+int > #int))*

The following defines Progol's definite mode language $\mathcal{L}(M)$.

Definition 41 (Definite mode language $\mathcal{L}(M)$) *Let C be a definite clause with a defined total ordering over the literals and M be a set of mode declarations. $C = h \leftarrow b_1, .., b_n$ is in the definite mode language $\mathcal{L}(M)$ if and only if 1) h is the atom of a modeh declaration in M with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 2) every atom b_i in the body of C is the atom of a modeb declaration in M with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 3) every variable of +type in any atom b_i is either of +type in h or of -type in some atom b_j, $1 \leq j < i$.*

Like Golem, Progol constructs clauses of bounded depth (see Definition 39).

[3] In practice this means a large number, e.g. Progol considers a maximum of 100 alternative solutions for the recall.

Definition 42 (Depth-bounded mode language $\mathcal{L}_i(M)$) *Let C be a definite clause with a defined total ordering over the literals and M be a set of mode declarations. C is in $\mathcal{L}_i(M)$ if and only if C is in $\mathcal{L}(M)$ and all variables in C have depth at most i according to Definition 39.*

Progol searches a bounded sub-lattice for each example e relative to background knowledge B and mode declarations M. The sub-lattice has a most general element which is the empty clause, $\square$, and a least general element $\bot_i$ which is the most specific element in $\mathcal{L}_i(M)$ such that

$$B \wedge \bot_i \wedge \neg e \vdash_h \square$$

where $\vdash_h \square$ denotes derivation of the empty clause in at most h resolutions. The following definition describes a bottom clause $\bot_i$ for a depth-bounded mode language $\mathcal{L}_i(M)$.

Definition 43 (Most-specific clause or bottom clause) *Let h and i be natural numbers, B be a set of Horn clauses, $e = a \leftarrow b_1, .., b_n$ be a definite clause, M be a set of mode declarations containing exactly one modeh m such that $a(m) \succeq a$ and $\hat{\bot}$ be the most-specific definite clause such that $B \wedge \hat{\bot} \wedge \neg e \vdash_h \square$. $\bot_i$ is the most-specific clause in $\mathcal{L}_i(M)$ such that $\bot_i \succeq \hat{\bot}$. C is the most-specific clause in $\mathcal{L}$ if for all C' in $\mathcal{L}$ we have $C' \succeq C$. $\overrightarrow{\bot}$ is $\bot_i$ with a defined ordering over the literals.*

In this thesis, we refer to $\bot_i$ as $\overrightarrow{\bot}$ or $\bot$ depending on whether we use the ordering of the literals or not. Progol's algorithm for constructing the bottom clause ($\bot_i$) is given in Appendix A.

Example 10 *Let M be the following mode declarations.*

modeh(,reverse(+list,-list))*	*modeb(*,+list=[-int|-list]*
modeb(,+any= #any)*	*modeb(*,reverse(+list,-list))*
modeb(,append(+list,[+int],-list))*	

Let background knowledge B be defined as follows.

$$B = \begin{cases} any(Term) \leftarrow \\ list([]) \leftarrow \\ list([H|T]) \leftarrow list(T) \\ Term = Term \leftarrow \\ reverse([],[]) \leftarrow \\ append([],X,X) \leftarrow \\ append([H|T],L1,[H|L2]) \leftarrow append(T,L1,L2) \end{cases}$$

Let $h = 30$ and $i = 3$ and let the example e be as below.

$$e = reverse([1],[1]) \leftarrow$$

In this case $\bot_i$ is as follows.

$$\begin{aligned} \bot_i = reverse(A,A) \quad \leftarrow \quad & A = [1], A = [B|C], B = 1, C = [], \\ & reverse(C,C), append(C,[B],A) \end{aligned}$$

Progol uses a refinement operator to search a hypothesis space bounded by the bottom clause. As described in [Mug95], the refinement operator in Progol is designed to maintain the relationship $\Box \succeq H \succeq \bot$ for each clause H and also to avoid or reduce redundancy. Since $H \succeq \bot$, it is the case that there exists a substitution θ such that $H\theta \subseteq \bot$. Thus for each literal l in H there exists a literal l' in $\bot$ such that $l\theta = l'$. Hence, there is a uniquely defined subset $\bot(H)$ consisting of all l' in $\bot$ for which there exists l in H and $l\theta = l'$. In order to choose an arbitrary subset S' of a set S an index k is maintained. For each value of k between 1 and n, the cardinality of S, it is decided whether to include the kth element of S in S'. The set of all series of n choices corresponds to the set of all subsets of S and for each subset of S there is exactly one series of n choices. Hence, Progol's refinement operator maintains both k and θ to avoid redundancy and maintain the relationship $\Box \succeq H \succeq \bot$

Definition 44 (Refinement operator ρ as defined in [Mug95]) *Let h, i, B, e, M and $\bot_i$ be defined as in Definition 43 and let n be the cardinality of $\bot_i$. Let k be a natural number, $1 \leq k \leq n$. Let C be a clause in $\mathcal{L}_i(M)$ and θ be a substitution such that $C\theta \subseteq \bot_i$. Below, a literal l corresponding to a mode m_l in M is denoted simply as $p(v_1, .., v_m)$ despite the sign of m_l and function symbols in $a(m_l)$. A variable is splittable if it corresponds to a +type or -type in a modeh or if it corresponds to a -type in a modeb. $\langle C', \theta', k' \rangle$ is in $\rho(\langle C, \theta, k \rangle)$ if and only if either*

1. *$C' = C \cup \{l\}$, $k' = k$, $\langle l, \theta' \rangle$ is in $\delta(\theta, k)$ and $C' \in \mathcal{L}_i(M)$ or*

2. *$C' = C$, $k' = k + 1$, $\theta' = \theta$ and $k < n$.*

where $\delta(\theta, k)$ is defined as follows. $\langle p(v_1, .., v_m), \theta' \rangle$ is in $\delta(\theta, k)$ if and only if θ' is initialised to θ, $l_k = p(u_1, .., u_m)$ is the kth literal of $\bot_i$ and for each j, $1 \leq j \leq m$,

1. *if u_j is splittable then $v_j/u_j \in \theta'$ else $v_j/u_j \in \theta$ or*

2. *if u_j is splittable then v_j is a new variable not in dom(θ) and $\theta' = \theta \cup \{v_j/u_j\}$.*

The refinement operator ρ defined in Definition 44 allows more than one literal in H to be mapped to the same literal l' in $\bot$ (i.e. $k' = k$). However, this is not allowed in Progol's implementation [4] for the sake of efficiency and index k is always incremented after each step. This means each literal of $\bot$ can be considered only once. In the following, we give a revised definition which describes the refinement operator as implemented in Progol. This also includes a corrected definition for function δ which uses the iterative construction of θ'.

Definition 45 (Refinement operator ρ as implemented in Progol[1]) *Let h, i, B, e, M and $\bot_i$ be defined as in Definition 43 and let n be the cardinality of $\bot_i$. Let k be a natural number, $1 \leq k \leq n$. Let C be a clause in $\mathcal{L}_i(M)$ and θ be a substitution such that $C\theta \subseteq \bot_i$. Below, a literal l corresponding to a mode m_l in M is denoted simply as $p(v_1, .., v_m)$ despite the sign of m_l and function symbols in $a(m_l)$. A variable is splittable if it corresponds to a +type or -type in a modeh or if it corresponds to a -type in a modeb. $\langle C', \theta', k' \rangle$ is in $\rho(\langle C, \theta, k \rangle)$ if and only if either*

1. *$C' = C \vee l$, $k' = k + 1$, $k < n$ and $\langle l, \theta' \rangle$ is in $\delta(\theta, k)$ and $C' \in \mathcal{L}_i(M)$ or*

2. *$C' = C$, $k' = k + 1$, $\theta' = \theta$ and $k < n$.*

where $\delta(\theta, k)$ is defined as follows. $\langle p(v_1, .., v_m), \theta'_m \rangle$ is in $\delta(\theta, k)$ if and only if $l_k = p(u_1, .., u_m)$ is the kth literal of $\bot_i$, $\theta'_0 = \theta$ and θ'_j for each j, $1 \leq j \leq m$ is defined as follows:

[4] E.g. Progol4.1 available from: http://www.doc.ic.ac.uk/~shm/Software/progol4.1/

1. if $v_j/u_j \in \theta'_{j-1}$ then $\theta'_j = \theta'_{j-1}$ or

2. if u_j is splittable then $\theta'_j = \theta'_{j-1} \cup \{v_j/u_j\}$ where v_j is a new variable not in $dom(\theta'_{j-1})$.

As noted in [Mug95], the variables in $\bot_i$ form a set of equivalence classes over the variables in any clause C which θ-subsumes $\bot_i$. Thus the equivalence class of u in θ could be written as $[v]_u$ representing the set of all variables in C such that v/u is in θ. The second choice in the definition of δ adds a new variable to an equivalence class $[v_j]_{u_j}$. This will be referred to as *splitting* the variable u_j. Note that in Definition 45 a variable is not splittable if it corresponds to a +type in a modeb since the resulting clause would violate the mode declaration language $\mathcal{L}(M)$ (see Definition 41). In some problems, given enough training examples, the target hypothesis can be learned without variable splitting (using only the variable bindings from the bottom clause). For this reason, the default refinement operators in some Progol-like ILP systems, including Aleph [Sri07], do not split variables and only consider adding literals from the bottom clause (i.e. the second choice of δ in Definition 45 is not implemented). However, it can be shown that there are problems where the target hypothesis cannot be found by a Progol-like ILP system without variable splitting. The following is an example where variable splitting is needed.

Example 11 (variable splitting) *Consider learning a half adder logical circuit that performs an addition operation on two binary digits and produces a sum and a carry value which are both binary digits. Suppose M consists of the following mode declarations:*

modeh(1, add(+bin, +bin, -bin, -bin))
modeb(1, xor(+bin, +bin, -bin))
modeb(1, and(+bin, +bin, -bin))

The type and other background knowledge are defined as follows:

$$B = \left\{ \begin{array}{lll} bin(0) \leftarrow, & bin(1) \leftarrow, & and(0,0,0) \leftarrow, \\ and(0,1,0) \leftarrow, & and(1,0,0) \leftarrow, & and(1,1,1) \leftarrow, \\ xor(0,0,0) \leftarrow, & xor(0,1,1) \leftarrow, & xor(1,0,1) \leftarrow, \\ xor(1,1,0) \leftarrow & & \end{array} \right.$$

The positive and negative examples are as follows:

$$E = \begin{cases} add(1,0,1,0) \leftarrow, & add(0,0,0,0) \leftarrow, & add(0,1,1,0) \leftarrow, \\ add(1,1,0,1) \leftarrow, & \leftarrow add(0,1,0,1), & \leftarrow add(1,0,0,1), \\ \leftarrow add(1,1,1,0), & \leftarrow add(1,1,1,1), & \leftarrow add(0,1,1,1) \end{cases}$$

Let $h = 30$ and $i = 3$ and let the first positive example be $e = add(1,0,1,0) \leftarrow$. In this case $\perp_i$ is as follows:

$$\begin{aligned} \perp_i \;=\; & add(A,B,A,B) \leftarrow xor(A,A,B), xor(A,B,A), xor(B,A,A), \\ & xor(B,B,B), and(A,A,A), and(A,B,B), and(B,A,B), \\ & and(B,B,B) \end{aligned}$$

Using the refinement operator in Definition 45, Progol can learn the following target hypothesis:

$$add(A,B,C,D) \leftarrow xor(A,B,C), and(A,B,D)$$

However, this clause cannot be generated from $\perp_i$ without variable splitting, because the bottom clause contains only two distinct variables and yet the target clause contains four variables. $\diamond$

This example represents a group of problems which cannot be learned by a Progol-like ILP system without variable splitting. In general, if the target clause includes a predicate with more than two variables which are defined over a binary domain, then in the bottom clause at least two arguments of this predicate always represent the same variable. This then requires variable splitting in order to generate the target clause from the bottom clause. Progol's refinement operator uses variable splitting by default, however it can be turned off by the user. Aleph's default refinement operator does not implement variable splitting and only considers adding literals from the bottom clause. Variable splitting in Aleph is implemented in an optional setting where equality literals between variables are inserted into the bottom clause to maintain equivalence classes over the variables. However, introducing equality literals in the bottom clause increases the search space considerably and can make the search explore redundant clauses. According to Aleph's manual [Sri07], if variable splitting is turned on (*splitvars* is set to true) the bottom clause can be extremely large and probably not practical for large numbers of variable co-references. For the problem mentioned in Example 11, Aleph can find the correct target hypothesis if *splitvars* is set to true. However, in this case Aleph considers a bottom clause with 101 literals compared with the bottom clause with 9 literals considered by Progol.

In this section we show that Progol's refinement cannot be described by the general subsumption order and that we need the notion of "sequential subsumption" in order to characterise Progol's refinement space. It can be shown that a refinement operator cannot be both complete and non-redundant [NCdW97]. However, a refinement operator can be weakly complete and non-redundant (optimal). As mentioned in the previous section, Progol's ρ is designed to be non-redundant and therefore it cannot be complete. However, it is known [Mug95] that Progol's refinement operator is also not weakly complete with respect to the general subsumption order as demonstrated in Example 7. In this example, clause C is in $\mathcal{L}$, but given the ordering over $\perp$ there will be no element of Progol's $\rho^*(\Box)$ containing this clause or a subsume-equivalent of this clause.

The authors of [BS99] describe two types of Progol's incompleteness. Example 7 is related to the first type of incompleteness which is due to the choice of ordering in the bottom clause and the variable dependencies in the literals. As mentioned in the previous section, Progol's refinement uses an indexing over the literals and the literals in $\perp$ can only be considered from left to right. Moreover, each literal from $\perp$ can be selected only once. This leads to the second type of incompleteness. The example below shows that Progol's refinement space is not a lattice with respect to the general subsumption, as the least general generalisation of clauses is not always in the refinement space.

Example 12 *Let C, D and $\perp$ be clauses as defined below.*

$$\begin{aligned} C &= p(X,Y) \leftarrow q(X,X), q(Y,W). \\ D &= p(X,Y) \leftarrow q(Z,X), q(Y,Y). \\ \perp &= p(X,Y) \leftarrow q(X,X), q(Y,Y). \end{aligned}$$

C and D can be generated by Progol's refinement (i.e. $C, D \in \rho^*(\Box)$), however, clause E below which is the least general generalisation (lgg) of C and D cannot be generated

(i.e. $E \notin \rho^*(\Box)$).

$$E = p(X,Y) \leftarrow q(Z,X), q(U,U), q(Y,W).$$

$\diamond$

Example 12 is related to the second type of incompleteness which is due to the fact that each literal from $\perp$ can be selected only once. Clause E, therefore, cannot be in $\rho^*(\Box)$ as this will require more than one literal of E to be mapped to the same literal of $\perp$. As another example of the second type of incompleteness, consider the following example adapted from [BS99].

Example 13 *Let $\perp = p(X) \leftarrow q(X,X)$, then Progol's refinement only considers the following hypotheses.*

$$\begin{array}{rcl} C & = & p(X) \\ D & = & p(X) \leftarrow q(X,Y) \\ E & = & p(X) \leftarrow q(X,X) \end{array}$$

However, the following clauses which subsume $\perp$ are not considered by Progol's refinement:

$$\begin{array}{rcl} C'_1 & = & p(X) \leftarrow q(X,Y), q(Y,X) \\ C'_2 & = & p(X) \leftarrow q(X,Y), q(Y,Z), q(Z,X) \\ C'_3 & = & p(X) \leftarrow q(X,Y), q(Y,Z), q(Z,W), q(W,X) \\ \dots & & \end{array}$$

$\diamond$

In this example clause C'_n can be constructed only if more than one literal (i.e. $n+1$ literals) from C'_n could be mapped to the same literal q(X,X) from $\perp$ (which is not allowed in Progol's refinement). This example demonstrates an incompleteness with respect to θ-subsumption. However, as shown in Section 3.1 the missing subsumption relations are also related to infinite ascending chains which is an undesirable property of θ-subsumption

$$\begin{tikzcd}
C: & & p(x,y) \arrow[dl, dashed] \arrow[d] \arrow[dr] & & \\
\rho(C): & p(x,x) \arrow[d, dashed] & p(x,y) \leftarrow q(x,z) \arrow[dl, dashed] \arrow[d, dashed, red] \arrow[dr] & p(x,y) \leftarrow r(w,y) \arrow[dr, dashed, red] & \\
\rho^2(C): & p(x,x) \leftarrow q(x,z) \arrow[d, dashed] & \begin{array}{c} p(x,y) \leftarrow q(x,z), \\ q(z,w) \end{array} \arrow[d, dashed, red] & p(x,y) \leftarrow q(x,x) \arrow[d] & p(x,y) \leftarrow r(y,y) \arrow[d, dashed, red] \\
\rho^3(C): & p(x,x) \leftarrow q(x,y) \arrow[d, dashed] & \begin{array}{c} p(x,y) \leftarrow q(x,z), \\ q(z,x) \end{array} \arrow[d, dashed, red] & \begin{array}{c} p(x,y) \leftarrow q(x,x), \\ r(w,y) \end{array} \arrow[ddl] & \begin{array}{c} p(x,y) \leftarrow r(y,y), \\ q(x,z) \end{array} \arrow[ddll, dashed, red] \\
\rho^n(C): & \ldots & \ldots \arrow[d, dashed, red] & & \\
\perp: & & \begin{array}{c} p(x,y) \leftarrow q(x,x), \\ r(y,y) \end{array} & &
\end{tikzcd}$$

Figure 3.3: Part of a refinement graph bounded by a bottom clause as in Progol. Dashed lines represent refinement steps which are not considered by Progol's refinement. Red dashed lines represent missing refinement steps which lead to incompleteness with respect to general subsumption

Figure 3.3 summarises the two types of incompleteness discussed in this section. This figure shows part of a refinement graph bounded by a bottom clause as in Progol. Suppose that the bottom clause $\perp$ is given by $p(x,y) \leftarrow q(x,x), r(y,y)$. Dashed lines represent refinement steps which are not considered by Progol's refinement. For example, no refinement step from $p(x,y)$ to $p(x,x)$ is considered because according to Progol's refinement operator, a literal is not allowed to be more specific than the corresponding literal from the bottom clause (i.e. $p(x,y)$ in this case). Red (grey in black and white) dashed lines represent missing refinement steps which lead to incompleteness with respect to general subsumption. For example, no refinement step from $p(x,y) \leftarrow r(w,y)$ to $p(x,y) \leftarrow r(w,y), q(x,z)$ is considered by Progol's refinement due to the choice of ordering in the bottom clause. As shown in example 7, the choice of ordering in the bottom clause and the variable dependencies in the literals could lead to incompleteness (first type of incompleteness). Moreover, the refinement step from $p(x,y) \leftarrow q(x,z)$ to $p(x,y) \leftarrow q(x,z), q(z,x)$ is missing because each literal from the bottom clause can be selected only once (second type of incompleteness).

As mentioned before, Progol's refinement operator scans $\perp$ from left to right and for each literal l' of $\perp$ decides whether to include a generalisation of it (i.e. l, where $l\theta = l'$) in H or not. $H\theta$ can be, therefore, characterised as a "subsequence" of $\perp$ rather than a "subset" of $\perp$. In the following sections we first define a special case of subsumption based on the idea of subsequences, and then we show how Progol's refinement can be characterised using sequential subsumption.

3.3 Ordered clauses and sequential subsumption

In this section we define the concepts of ordered clauses and sequential subsumption which will be used for characterising clause refinement in a Progol-like ILP system. According to Definition 41, clauses which are considered by Progol's refinement (i.e. clauses in $\mathcal{L}(M)$) are defined with a total ordering over the literals. In order to characterise Progol's refinement we adapt an explicit representation for ordered clauses. The concept of ordered clauses has been used before in ILP. For example, when defining upward refinement operators it is sometime necessary to duplicate literals in order to correctly invert an elementary substitution. Duplication of literals is not allowed in the standard representation of clauses (which use a set notation) and therefore ordered clauses are used instead [NCdW97]. A subsumption relation for ordered clauses is studied in [KOHH06]. The difference between this subsumption order and the subsumption order considered in this thesis is discussed in Section 3.5. There are also other applications of ordered clauses and sequential subsumption, for example in the context of data mining from sequential data (e.g. [LD04]). In this thesis we use the same notion used in [NCdW97] and an ordered clause is represented as a disjunction of literals (i.e. $L_1 \vee L_2 \vee \cdots \vee L_n$). The set notation (i.e. $\{L_1, L_2, \ldots, L_n\}$) is used to represent conventional clauses.

Definition 46 (Ordered clause) *An ordered clause $\overrightarrow{C}$ is a sequence of literals $L_1, L_2, \ldots, L_n$ and denoted by $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$. The set of literals in $\overrightarrow{C}$ is denoted by C.*

Unlike conventional clauses, the order and duplication of literals matter for ordered clauses. For example, $\overrightarrow{C} = p(X) \vee \neg q(X)$, $\overrightarrow{D} = \neg q(X) \vee p(X)$ and $\overrightarrow{E} = p(X) \vee \neg q(X) \vee$

$p(X)$ are different ordered clauses while they all correspond to the same conventional clause, i.e. $C = D = E = \{p(X), \neg q(X)\}$.

Selection of two clauses is defined as a pair of compatible literals and this concept was used by Plotkin to define least generalisation for clauses [Plo71]. However, in this thesis we use selections to define mappings of literals between two ordered clauses.

Definition 47 (Compatible literals) *Literals L and M are compatible if they have the same sign and predicate symbol.*

Definition 48 (Selection of clauses) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$ be ordered clauses. A selection of $\overrightarrow{C}$ and $\overrightarrow{D}$ is a pair (i, j) where L_i and M_j are compatible literals.*

Definition 49 (Selection function) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$ be ordered clauses. A set s of selections of $\overrightarrow{C}$ and $\overrightarrow{D}$ is called a selection function if it is a total function of $\{1, 2, \ldots, n\}$ into $\{1, 2, \ldots, m\}$.*

Example 14 *Let $\overrightarrow{C} = L_1 \vee L_2 \vee L_3$ and $\overrightarrow{D} = M_1 \vee M_2 \vee M_3 \vee M_4$ be two ordered clauses and the set of all selections of $\overrightarrow{C}$ and $\overrightarrow{D}$ be $S = \{$(1,1), (1,2), (2,1), (2,2), (3,4)$\}$. Then, $s_1 = \{$(1,1), (2,2), (3,4)$\}$, $s_2 = \{$(1,1), (2,1), (3,4)$\}$ and $s_3 = \{$(1,2), (2,1), (3,4)$\}$ are examples of selection functions of $\overrightarrow{C}$ and $\overrightarrow{D}$.* $\diamond$

Definition 50 (Subsequence) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_l$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$ be ordered clauses. $\overrightarrow{C}$ is a subsequence of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \sqsubseteq \overrightarrow{D}$, if there exists a strictly increasing selection function $s \subseteq \{1, \ldots, l\} \times \{1, \ldots, m\}$ such that for each $(i, j) \in s$, $L_i = M_j$.*

Example 15 *In Figure 3.4, $\overrightarrow{C}$ is a subsequence of $\overrightarrow{B}$ because there exists an increasing selection function $s_1 = \{(1,1), (2,3), (3,4)\}$ which maps literals from $\overrightarrow{C}$ to equivalent literals from $\overrightarrow{B}$. However, $\overrightarrow{D}$ is not a subsequence of $\overrightarrow{B}$ because an increasing selection function does not exist for $\overrightarrow{D}$ and $\overrightarrow{B}$.* $\diamond$

Definition 51 (Ordered substitution) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_l$ be an ordered clause and θ be a substitution. $\overrightarrow{C}\theta$ is defined as follows, $\overrightarrow{C}\theta = L_1\theta \vee L_2\theta \vee \cdots \vee L_l\theta$.*

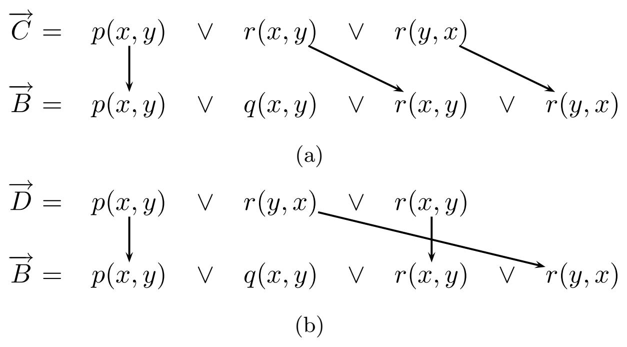

$\overrightarrow{C} = p(x,y) \vee r(x,y) \vee r(y,x)$
$\overrightarrow{B} = p(x,y) \vee q(x,y) \vee r(x,y) \vee r(y,x)$
(a)
$\overrightarrow{D} = p(x,y) \vee r(y,x) \vee r(x,y)$
$\overrightarrow{B} = p(x,y) \vee q(x,y) \vee r(x,y) \vee r(y,x)$
(b)

Figure 3.4: (a) $\vec{C}$ is a subsequence of $\vec{B}$ because there exists a strictly increasing selection function $s_1 = \{(1,1),(2,3),(3,4)\}$ which maps each literal from $\vec{C}$ to an equivalent literal from $\vec{B}$ (b) $\vec{D}$ is not a subsequence of $\vec{B}$.

Definition 52 (Sequential subsumption) *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses. We say $\overrightarrow{C}$ is a sequential generalisation of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \succeq_s \overrightarrow{D}$, if there exists a substitution θ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. $\overrightarrow{C}$ is a proper sequential generalisation of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \succ_s \overrightarrow{D}$, if $\overrightarrow{C} \succeq_s \overrightarrow{D}$ and $\overrightarrow{D} \not\succeq_s \overrightarrow{C}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ are equivalent with respect to sequential subsumption, denoted by $\overrightarrow{C} \sim_s \overrightarrow{D}$, if $\overrightarrow{C} \succeq_s \overrightarrow{D}$ and $\overrightarrow{D} \succeq_s \overrightarrow{C}$.*

Example 16 *Let $\overrightarrow{B} = p(X_1,Y_1) \vee q(X_1,Y_1) \vee r(X_1,Y_1) \vee r(Y_1,X_1)$, $\overrightarrow{C} = p(X_2,Y_2) \vee r(U_2,Y_2) \vee r(Y_2,V_2)$ and $\overrightarrow{D} = p(X_3,Y_3) \vee r(Y_3,V_3) \vee r(U_3,Y_3)$ be ordered clauses. Let $\theta_1 = \{X_2/X_1, Y_2/Y_1, U_2/X_1, V_2/X_1\}$, then $\overrightarrow{C}\theta_1$ is a subsequence of $\overrightarrow{B}$ and therefore $\overrightarrow{C} \succeq_s \overrightarrow{B}$. However, there is no substitution θ_2 such that $\overrightarrow{D}\theta_2$ is a subsequence of $\overrightarrow{B}$ and therefore $\overrightarrow{D} \not\succeq_s \overrightarrow{B}$. Note that for conventional clauses B, C and D we have $C\theta_1 \subseteq B$ and similarly for $\theta_2 = \{X_3/X_1, Y_3/Y_1, V_3/X_1, U_3/X_1\}$ we have $D\theta_2 \subseteq B$ and therefore $C \succeq B$ and $D \succeq B$.* $\diamond$

The following theorem shows the relationship between sequential subsumption and the general subsumption order.

Theorem 1 *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses. If $\overrightarrow{C} \succeq_s \overrightarrow{D}$, then $C \succeq D$.*

Proof. Suppose $\overrightarrow{C} \succeq_s \overrightarrow{D}$, then according to Definition 52 there exists a substitution θ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. Let $\overrightarrow{C}\theta = L_1\theta \vee L_2\theta \vee \cdots \vee L_l\theta$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$. Then for every literal $L_i\theta$ in $\overrightarrow{C}\theta$ there exists a literal M_j in $\overrightarrow{D}$ such that $L_i\theta = M_j$, and therefore $C\theta \subseteq D$. Hence, $C \succeq D$. $\square$

Ordered clauses	**Conventional clauses**
$\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$	$C = \{L_1, L_2, \ldots, L_n\}$
Mapping of literals (s)	Undefined
Subsequence ($\sqsubseteq$)	Subset ($\subseteq$)
Sequential subsumption ($\succeq_s$)	Subsumption ($\succeq$)

Table 3.2: A comparison between ordered clauses and conventional clauses.

Note that as shown in Example 16, the converse of Theorem 1 does not hold in general. Table 3.3 shows a comparison between corresponding concepts for ordered clauses and conventional clauses.

3.4 Subsumption relative to a bottom clause

As shown in Section 3.2, clause refinement in Progol-like ILP systems is incomplete with respect to the general subsumption order and it cannot be properly described by the general subsumption order. In this section we define a subsumption order relative to $\bot$ (i.e. $\succeq_\bot$) which can capture clause refinement in these systems. First we define $\overrightarrow{\mathcal{L}}_\bot$ as the set of definite ordered clauses which are sequential generalisation of $\overrightarrow{\bot}$.

Definition 53 **($\overrightarrow{\mathcal{L}}_\bot$)** *Let $\overrightarrow{\bot}$ be a bottom clause as defined in Definition 43 and $\overrightarrow{C}$ a definite ordered clause. $\overrightarrow{\bot_v}$ is $\overrightarrow{\bot}$ with all variable positions populated with new and distinct variables. Let θ_v be a variable substitution such that $\overrightarrow{\bot_v}\theta_v = \overrightarrow{\bot}$. $\overrightarrow{C}$ is in $\overrightarrow{\mathcal{L}}_\bot$ if $\overrightarrow{C}\theta_v$ is a subsequence of $\overrightarrow{\bot}$.*

Example 17 *Let $\overrightarrow{\bot} = p(X) \leftarrow q(X), r(X), s(X,Y), s(Y,X)$ and according to Definition 53, we have $\overrightarrow{\bot_v} = p(V_1) \leftarrow q(V_2), r(V_3), s(V_4,V_5), s(V_6,V_7)$ and $\theta_v = \{V_1/X, V_2/X, V_3/X, V_4/X, V_5/Y, V_6/Y, V_7/X\}$. Then $\overrightarrow{C} = p(V_1) \leftarrow r(V_2), s(V_6,V_7)$, $\overrightarrow{D} = p(V_1) \leftarrow r(V_1), s(V_6,V_1)$ and $\overrightarrow{E} = p(V_1) \leftarrow r(V_1), s(V_4,V_5)$ are in $\overrightarrow{\mathcal{L}}_\bot$ as $\overrightarrow{C}\theta_v$, $\overrightarrow{D}\theta_v$ and $\overrightarrow{E}\theta_v$ are subsequences of $\overrightarrow{\bot}$.* $\diamond$

In this section we show that the refinement space of a Progol-like ILP system can be characterised using $\overrightarrow{\mathcal{L}}_\bot$. We also define a subsumption order relative to a bottom clause which can be used to study Progol's refinement. According to the definition of

Progol's refinement operator (Definition 45), refinements of a clause are constructed by adding literals which are generalisations of a literal from $\overrightarrow{\bot}$. These literals are generated by δ (Definition 45) and they correspond to literal l_k from $\overrightarrow{\bot}$. A literal L_i from $\overrightarrow{C}$ is comparable (with respect to Progol's refinement) to a literal M_j from $\overrightarrow{D}$ if L_i and M_j are both mapped to the same literal of $\overrightarrow{\bot}$. This has been demonstrated in the following example.

Example 18 *Let $\overrightarrow{\bot}$, $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be as in Example 17 and as shown in Figure 3.1. The literals of $\overrightarrow{C}$ are mapped to the first, the third and the fifth literals of $\overrightarrow{\bot}$ respectively, and similarly the literals of $\overrightarrow{D}$ are mapped to the first, the second, the third and the fifth literals of $\overrightarrow{\bot}$ as in Figure 3.1.a. In this case, literals from $\overrightarrow{C}$ are comparable with the first, the third and the fourth literals from $\overrightarrow{D}$ respectively. However, in Figure 3.1.b, the third literal from $\overrightarrow{C}$ and the third literal from $\overrightarrow{E}$ are mapped to different literals from $\overrightarrow{\bot}$ and therefore they are not comparable with respect to Progol's refinement (though they are comparable with respect to general subsumption or sequential subsumption).* $\diamond$

In the following we define a subsumption order relative to $\bot$ (i.e. $\succeq_\bot$) in which the mapping between literals with respect to the bottom clause is considered. First we define variable substitutions relative to $\bot$.

Definition 54 (Substitution relative to $\bot$) *Let $\overrightarrow{\bot}$ be the bottom clause as defined in Definition 43 and θ_v be as defined in Definition 53. Let $\theta_\bot = \{v_j/v_i | \{v_i/u, v_j/u\} \subseteq \theta_v$ and $i < j\}$, then $\theta_\bot$ is a substitution relative to $\bot$.*

Definition 55 (Subsumption relative to $\bot$) *Let $\overrightarrow{\bot}$ be a bottom clause as defined in Definition 43, $\overrightarrow{\mathcal{L}}_\bot$ be as defined in Definition 53 and $\theta_\bot$ be as defined in Definition 54. Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$. We say $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\bot$, denoted by $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$, if there exists a substitution $\theta \subseteq \theta_\bot$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. $\overrightarrow{C}$ is a proper generalisation of $\overrightarrow{D}$ relative to $\bot$, denoted by $\overrightarrow{C} \succ_\bot \overrightarrow{D}$, if $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$ and $\overrightarrow{D} \not\succeq_\bot \overrightarrow{C}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ are equivalent with respect to subsumption relative to $\bot$, denoted by $\overrightarrow{C} \sim_\bot \overrightarrow{D}$, if $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$ and $\overrightarrow{D} \succeq_\bot \overrightarrow{C}$.*

Example 19 *Let $\overrightarrow{\bot}$, θ_v, $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be as in Example 17. Then, according to Definition 54, $\theta_\bot = \{V_2/V_1, V_3/V_1, V_4/V_1, V_7/V_1, V_3/V_2, V_4/V_2, V_7/V_2, V_7/V_3, V_7/V_4, V_6/V_5\}$.*

Then, $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\bot$ since there is a substitution $\theta = \{V_2/V_1, V_7/V_1\} \subseteq \theta_\bot$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. However, $\overrightarrow{C}$ does not subsume $\overrightarrow{E}$ relative to $\bot$ since there is no substitution $\theta \subseteq \theta_\bot$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{E}$. Note that $\overrightarrow{C}$ subsumes $\overrightarrow{E}$ with respect to θ-subsumption (see Figure 3.1). $\diamond$

In the following we first define $\overrightarrow{\mathcal{L}}_\bot(M)$ by analogy to Progol's $\mathcal{L}_i(M)$ and then we re-define Progol's refinement operator for ordered clauses in $\overrightarrow{\mathcal{L}}_\bot(M)$.

Definition 56 ($\overrightarrow{\mathcal{L}}_\bot(M)$) *Let $\overrightarrow{\mathcal{L}}_\bot$ and $\mathcal{L}_i(M)$ be as defined in Definition 53 and Definition 42 respectively. $\overrightarrow{C}$ is in $\overrightarrow{\mathcal{L}}_\bot(M)$ if and only if $\overrightarrow{C}$ is in $\overrightarrow{\mathcal{L}}_\bot$ and C is in $\mathcal{L}_i(M)$.*

Note that according to Definitions 41 and 42 a total ordering has been assumed over the literals of the clauses in $\mathcal{L}(M)$ and $\mathcal{L}_i(M)$. This ordering is explicitly defined for the ordered clauses in $\overrightarrow{\mathcal{L}}_\bot(M)$. In Definition 45, the refinement operator ρ is defined for clauses in $\mathcal{L}_i(M)$. However, ρ can also be defined for clauses in $\overrightarrow{\mathcal{L}}_\bot(M)$ if we let C and C' be ordered clauses in $\overrightarrow{\mathcal{L}}_\bot(M)$ and $\overrightarrow{C}\theta$ be a subsequence (rather than a subset) of the bottom clause.

Definition 57 (Refinement operator ρ for ordered clauses) *Let h, i, B, e, M and $\overrightarrow{\bot}$ be defined as in Definition 43 and let n be the cardinality of $\overrightarrow{\bot}$. Let k be a natural number, $1 \leq k \leq n$. Let $\overrightarrow{C}$ be a clause in $\overrightarrow{\mathcal{L}}_\bot(M)$ and θ be a substitution such that $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{\bot}$. Below, a literal l corresponding to a mode m_l in M is denoted simply as $p(v_1, .., v_m)$ despite the sign of m_l and function symbols in $a(m_l)$. A variable is splittable if it corresponds to a +type or -type in a modeh or if it corresponds to a -type in a modeb. $\langle \overrightarrow{C'}, \theta', k'\rangle$ is in $\rho(\langle \overrightarrow{C}, \theta, k\rangle)$ if and only if either*

1. *$\overrightarrow{C'} = \overrightarrow{C} \vee l$, $k' = k + 1$, $k < n$ and $\langle l, \theta'\rangle$ is in $\delta(\theta, k)$ and $\overrightarrow{C'} \in \overrightarrow{\mathcal{L}}_\bot(M)$ or*
2. *$\overrightarrow{C'} = \overrightarrow{C}$, $k' = k + 1$, $\theta' = \theta$ and $k < n$.*

where $\delta(\theta, k)$ is defined as follows. $\langle p(v_1, .., v_m), \theta'_m\rangle$ is in $\delta(\theta, k)$ if and only if $l_k = p(u_1, .., u_m)$ is the kth literal of $\overrightarrow{\bot}$, $\theta'_0 = \theta$ and θ'_j for each j, $1 \leq j \leq m$ is defined as follows:

1. *if $v_j/u_j \in \theta'_{j-1}$ then $\theta'_j = \theta'_{j-1}$ or*

2. if u_j is splittable then $\theta'_j = \theta'_{j-1} \cup \{v_j/u_j\}$ where v_j is a new variable not in $dom(\theta'_{j-1})$.

Note that in Definition 57, $\overrightarrow{C}$ and $\overrightarrow{C'}$ are in $\overrightarrow{\mathcal{L}}_{\bot}(M)$, $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{\bot}$ and θ and θ' are subsets of θ_v such that for each $v_j/u_j \in \theta$ (or $v_j/u_j \in \theta'$), if v_j is a new and distinct variable then u_j must be a splittable variable.

As shown by the examples from Section 3.2, in particular Example 7, Progol's refinement cannot be complete or even weakly complete for general subsumption order. In the following we show that Progol's refinement can be weakly complete for $\langle \overrightarrow{\mathcal{L}}_{\bot}(M), \succeq_{\bot} \rangle$.

Lemma 1 *Let $\delta(\theta, k)$ be as defined in Definition 57 and $\overrightarrow{C}$ and $\overrightarrow{C'} = \overrightarrow{C} \vee l$ be ordered clauses in $\overrightarrow{\mathcal{L}}_{\bot}(M)$ such that $\overrightarrow{C}\theta$ and $\overrightarrow{C'}\theta'$ are subsequences of $\overrightarrow{\bot}$ and $l\theta' = l_k$ where l_k is the kth literal of $\overrightarrow{\bot}$. Then, there exists $\langle l', \theta'' \rangle$ in $\delta(\theta, k)$ such that l and l' are variants.*

Proof. Let literals l_k, l and l' be denoted simply by $p(u_1, .., u_m)$, $p(v_1, .., v_m)$ and $p(v'_1, .., v'_m)$ respectively, despite the sign and function symbols (as in Definition 57) i.e. $l_k = p(u_1, .., u_m)$, $l = p(v_1, .., v_m)$ and $l' = p(v'_1, .., v'_m)$. We have $p(v_1, .., v_m)\theta' = p(u_1, .., u_m)$. We show that $\langle l', \theta'' \rangle$ can be constructed using $\delta(\theta, k)$ and there exist variables $v'_1, .., v'_m$ and substitution θ'' such that $p(v'_1, .., v'_m)\theta'' = p(u_1, .., u_m)$ and l and l' are variants. Let $\theta''_0 = \theta$ and for each $v_j/u_j \in \theta'$ where $1 \leq j \leq m$ if v_j is a new variable with respect to $\{v_1, \ldots, v_{j-1}\}$ and u_j is splittable then, using choice 2 in the definition of δ, $\theta''_j = \theta''_{j-1} \cup \{v'_j/u_j\}$ where v'_j is a new variable not in $dom(\theta''_{j-1})$. Otherwise, by using choice 1 in the definition of δ, $\theta''_j = \theta''_{j-1}$. By construction, $\langle l', \theta''_m \rangle$ is in $\delta(\theta, k)$ and there is a one-to-one mapping between variables v'_j and v_j for $1 \leq j \leq m$. Variable substitutions $\sigma_1 = \{v_1/v'_1, \ldots, v_m/v'_m\}$ and $\sigma_2 = \{v'_1/v_1, \ldots, v'_m/v_m\}$ are therefore variable renamings. Hence, $l\sigma_1 = l'$ and $l'\sigma_2 = l$ and therefore l and l' are variants. □

Theorem 2 *Let ρ be as defined in Definition 57. Then ρ is weakly complete for $\langle \overrightarrow{\mathcal{L}}_{\bot}(M), \succeq_{\bot} \rangle$.*

Proof. We need to show that $\rho^*(\langle\Box, \emptyset, 1\rangle) = \overrightarrow{\mathcal{L}}_\perp(M)$. We show that for each $\overrightarrow{C} \in \overrightarrow{\mathcal{L}}_\perp(M)$, there exists a ρ-chain from $\Box$ to $\overrightarrow{C'}$ where $\overrightarrow{C'}$ and $\overrightarrow{C}$ are alphabetical variants. The proof is by induction on i, the number of literals in $\overrightarrow{C}$. If $i = 0$ then $\overrightarrow{C} = \Box$, and the empty chain satisfies the theorem. Assume for some j, $0 \leq j < i$, that the theorem is true, we will show that it is also true for $j+1$. Suppose the theorem is true for j, this implies that there is a ρ-chain from $\Box$ to an alphabetical variant of $\overrightarrow{C}_j$ such that $\overrightarrow{C}_j$ is an ordered clause in $\overrightarrow{\mathcal{L}}_\perp(M)$ with j literals added from $\overrightarrow{C}$. Therefore, there is a substitution θ such that $\overrightarrow{C}_j\theta$ is a subsequence of $\overrightarrow{\perp}$ and we assume that the j-th literal of $\overrightarrow{C}_j$ is mapped to the k-th literal of $\overrightarrow{\perp}$. Let $\overrightarrow{C}_{j+1} = \overrightarrow{C}_j \vee l$, where l is the leftmost literal of $\overrightarrow{C}$ which is not in $\overrightarrow{C}_j$ and l is mapped to the k'-th literal of $\overrightarrow{\perp}$, where $k < k'$ (because $\overrightarrow{C}_j$ and $\overrightarrow{C}_{j+1}$ are sequential generalisations of $\overrightarrow{\perp}$). Then there exists a ρ-chain from $\langle\overrightarrow{C}_j, \theta, k\rangle$ to $\langle\overrightarrow{C}_j, \theta, k'\rangle$ by repeatedly selecting choice 2 in the definition of ρ in order to skip $k' - k$ literals of $\overrightarrow{\perp}$. According to Lemma 1, there exists $\langle l', \theta'\rangle$ in $\delta(\theta, k')$ such that l and l' are variants. Therefore, by selecting choice 1 in the definition of ρ, $\overrightarrow{C'}_{j+1} = \overrightarrow{C}_j \vee l'$ is a variant of $\overrightarrow{C}_{j+1} = \overrightarrow{C}_j \vee l$, where $\langle\overrightarrow{C'}_{j+1}, \theta', k'+1\rangle \in \rho(\langle\overrightarrow{C}_j, \theta, k'\rangle)$. Thus, there is a ρ-chain from $\Box$ to a variant of $\overrightarrow{C}_{j+1}$ and this completes the proof. $\Box$

3.5 Related work and discussion

Progol's refinement operator and its incompleteness with respect to the general subsumption order were initially discussed in [Mug95]. The purpose of the present chapter was to characterise Progol's refinement space and to give an analysis of refinement operators for this space. In a previous attempt, the authors of [BS99] suggested weak subsumption for characterising Progol's refinement space. However, as we have shown in this chapter, weak subsumption cannot capture all aspects of Progol's refinement. Note that sequential subsumption implies weak subsumption. This is because if a selection function is strictly increasing then the injectivity property holds which, in turns, entails weak subsumption.

The sequential subsumption described in this chapter is similar to 'ordered subsumption' [KOHH06] and is defined based on the concept of subsequence. However, the subsequence relation considered in [KOHH06], assumes a mapping function which is monotonically increasing (rather than strictly increasing mapping function considered in our definition of subsequence). This means that in ordered subsumption, several literals from the first clause can be mapped to the same literal from the second clause. However, as already discussed in this chapter, in sequential subsumption (and subsumption relative to a bottom clause) distinct literals from the first clause can not be mapped to the same literal from the second clause. Hence, the results from this chapter are not applicable in the case of ordered subsumption.

As shown in this chapter, in order to characterise Progol's refinement, the sequential subsumption is not enough and we also need to consider the mapping between literals with respect to the bottom clause. For this reason we have defined a subsumption order relative to $\bot$ (i.e. $\succeq_\bot$) and we have shown that it can capture Progol's refinement. The framework used in [KOHH06] can be viewed as a general case for ordered and sequential subsumption. Whereas in this thesis we introduce subsumption order relative to a bottom clause.

In the context of data mining from sequential data, SeqLog [LD04, Lee06] is defined as a logical language for representing and mining sequential patterns in databases. Subsumption relations are also defined for simple and complex sequences and an optimal refinement operators is used in SeqLog's data-mining algorithms. A simple sequence is defined as a sequence of atoms whereas a complex sequence of atoms separated by operators which can be either 'direct successor' represented by $\lhd$ (which is omitted in writing) or '$<$' which is the transitive closure of $\lhd$. A simple sequence is therefore a degenerated form of a complex sequence in which all operators are $\lhd$. For example 'latex(FileName,tex) xdvi(FileName,tex) dvips(FileName,dvi)' is a simple sequence whereas 'latex(FileName,tex) $<$ dvips(FileName,dvi)' is a complex sequence which means atom dvips(FileName,dvi) occurs somewhere after latex(FileName,tex).

Based on simple and complex sequences, different forms of subsequences and subsumption relations are defined. Simple subsumption (s-subsumption) defines how a simple sequence subsumes a complex sequence. This is based on simple subsequences and very similar to the definition of sequential subsumption described in this chapter. However, general subsumption for SeqLog (SeqLog-subsumption) is more complicated and defines how a complex sequence subsumes a sequence. It is shown [Lee06] that time complexity of s-subsumption is polynomial but SeqLog-subsumption among complex sequences is NP-complete. It is also shown that lgg does not exist for any pair of simple sequences or complex sequences and s-subsumption and SeqLog-subsumption do not form lattices. However, optimal refinement operators are defined for simple and complex sequences. These optimal refinement operators are based on non-redundant operations which avoid duplicates by imposing an ordering over elementary (downward) refinement operations.

The ordered clause used in this thesis is similar to simple sequence in SeqLog and our sequential subsumption is comparable with s-subsumption. However, there is not any concept similar to complex sequences or SeqLog-subsumption in our approach and these could be regarded as the general forms of those used in our approach. On the other hand, no concept similar to bounded subsumption is defined for SeqLog. The SeqLog framework uses a constraint based mining together with the optimal refinement operators described above. SeqLog is also an attractive framework for ILP, e.g. because of optimal refinement operators. However, the SeqLog framework can be extended to a machine learning approach based on inverse entailment and in this case the results from bounded subsumption presented in this thesis can be used. These include efficient lgg-like operators which currently do not exist for SeqLog.

The subsumption order relative to a bottom clause can also be compared with the approaches which use some form of subsumption under object identity (e.g. [ELMS96, AR99]). The sequential subsumption defined in this chapter, introduces an injective mapping between the literals while the subsumption under object identity requires the injective mapping between the variables (objects). The subsumption under object identity, therefore, implies sequential subsumption. This is because the injection at object level entails the injection at literal level.

The subsumption under object identity is also related to the discussions in this chapter on implementation of variable splitting in bounded refinement operators. In the subsumption under object identity, it is assumed that each clause also includes a set of constraints in the body, e.g. in the form of inequalities between variables. This is similar to Aleph's approach for Variable splitting where equality literals between variables are inserted into the bottom clause to maintain equivalence. A (sequential) subsumption under object identity can be easily implemented by turning off the variable splitting in the bounded refinement operators described in this chapter, i.e. only selecting the first choice in the definition of ρ which adds literals from the bottom clause. With regards to Table 3.1, sequential subsumption under object identity will have the same advantages of bounded subsumption in terms of the existence of lgg and ideal refinement operators and linear time complexity. However, it will also have all four types of incompleteness described in Section 3.1, i.e. incompleteness types 0 to 3.

3.6 Summary

ILP systems which use some form of Inverse Entailment (IE) are based on clause refinement through a hypothesis space bounded by a most specific (bottom) clause. In this chapter we gave a new analysis of refinement in this setting. In particular, clause refinement in Progol's was revisited and discussed. We demonstrated that Progol's refinement is incomplete with respect to the general subsumption order and we discussed two different aspects of this incompleteness. Based on this analysis we introduced a subsumption order relative to a bottom clause and demonstrated how Progol's refinement can be characterised with respect to this order. This new subsumption order is based on the concepts of ordered clauses and sequential subsumption which we define in this thesis. This subsumption order, unlike previously suggested orders, characterises all aspects of Progol's refinement. We studied the properties of this subsumption order and showed that Progol's refinement is weakly complete for the subsumption order relative to a bottom clause.

Chapter 4

The lattice structure and morphisms of bounded subsumption

In this chapter we study the lattice structure and morphisms of the subsumption order relative to $\bot$. In Section 4.1 we show that $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ is a quasi-order and each pair of ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$ have a most general specialisation ($mgs_\bot$) and a least general generalisation ($lgg_\bot$) in $\overrightarrow{\mathcal{L}}_\bot$ and therefore $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ is a lattice. In this section we also describe two different type of downward covers for the lattice $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ and we show that for any given ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\bot$ such that $\overrightarrow{C} \succ_\bot \overrightarrow{D}$, there is a finite chain of downward covers from $\overrightarrow{C}$ to a variant of $\overrightarrow{D}$. In Section 4.2, we discuss a morphism between the lattice of bounded subsumption and an atomic lattice. We show that the lattice $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ isomorphic to an atomic lattice (i.e. $\langle \mathcal{A}_\bot, \succeq \rangle$). In Section 4.3 we show that the lattice $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ can also be mapped to a lattice of partitions. This mapping will be used in the next chapter for encoding the bounded subsumption lattice. Related work is discussed in Section 4.4. Section 4.5 summarises the chapter.

4.1 The lattice and cover structure of bounded subsumption

In the following we show that $\langle \overrightarrow{\mathcal{L}}_\bot, \succeq_\bot \rangle$ is a lattice. First we show that $\succeq_\bot$ is a quasi-order and then we prove that each pair of ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$ have a most general specialisation ($mgs_\bot$) and a least general generalisation ($lgg_\bot$) in $\overrightarrow{\mathcal{L}}_\bot$.

Lemma 2 *Subsequence relation ($\sqsubseteq$) is transitive.*

Proof. Let $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be ordered clauses such that $\overrightarrow{C} \sqsubseteq \overrightarrow{D}$ and $\overrightarrow{D} \sqsubseteq \overrightarrow{E}$. Then according to Definition 50 there exist strictly increasing selection functions s_4 and s_5 such that s_4 maps literals from $\overrightarrow{C}$ to equivalent literals from $\overrightarrow{D}$ and s_5 maps literals from $\overrightarrow{D}$ to equivalent literals from $\overrightarrow{E}$. Then $s = s_4 \circ s_5$ is also a strictly increasing function which maps literals from $\overrightarrow{C}$ to equivalent literals from $\overrightarrow{E}$ and therefore $\overrightarrow{C} \sqsubseteq \overrightarrow{E}$. $\square$

Theorem 3 *Subsumption order relative to $\bot$ ($\succeq_\bot$) is a quasi-order.*

Proof. Let $\overrightarrow{\bot}$ and $\overrightarrow{\mathcal{L}}_\bot$ be as defined in Definition 53 and $\theta_\bot$ be as defined in Definition 54. For every ordered clause $\overrightarrow{C}$ in $\overrightarrow{\mathcal{L}}_\bot$, we have $\overrightarrow{C} \succeq_\bot \overrightarrow{C}$. The relation $\succeq_\bot$ is therefore reflexive. Let $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$ such that $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$ and $\overrightarrow{D} \succeq_\bot \overrightarrow{E}$. Then there exist $\theta, \theta' \subseteq \theta_\bot$ such that $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{D}$ and $\overrightarrow{D}\theta' \sqsubseteq \overrightarrow{E}$. Thus, we have $\overrightarrow{C}\theta\theta' \sqsubseteq \overrightarrow{D}\theta'$ and according to Lemma 2, $\overrightarrow{C}\theta\theta' \sqsubseteq \overrightarrow{E}$. But $\theta\theta' \subseteq \theta_\bot$ and therefore according to Definition 55 $\overrightarrow{C} \succeq_\bot \overrightarrow{E}$. The relation $\succeq_\bot$ is reflexive and transitive and therefore it is a quasi-order. $\square$

In the following we prove that each pair of ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$ have *mgs* and *lgg*. As in [NCdW97] we use a sequence of pairs of compatible literals (i.e. selections) to bridge between the definitions of *mgs* and *lgg* for atoms and the definitions of *mgs* and *lgg* for clauses. The following definition is similar to Definition 14.23 in [NCdW97] adapted for subsumption relative to a bottom clause.

Definition 58 *Let $\overrightarrow{\bot}$, θ_v and $\overrightarrow{\mathcal{L}}_\bot$ be as defined in Definition 53, $\theta_\bot$ be as defined in Definition 54 and $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\bot$.$S = (L_1, M_1), \ldots, (L_n, M_n)$ is a sequence of pairs of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\overrightarrow{\bot}$ if $\overrightarrow{C_S} = L_1 \vee L_2 \vee \cdots \vee L_n$ is a subsequence of $\overrightarrow{C}$ and $\overrightarrow{D_S} = M_1 \vee M_2 \vee \cdots \vee M_n$ is a subsequence of $\overrightarrow{D}$ and $\overrightarrow{C_S}\theta_v = \overrightarrow{D_S}\theta_v$.*

Lemma 3 *Let $\overrightarrow{C}$, $\overrightarrow{D}$, S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ be as defined in Definition 58. Then, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ are in $\overrightarrow{\mathcal{L}}_\bot$.*

Proof. $\overrightarrow{C}$ and $\overrightarrow{D}$ are ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ and therefore $\overrightarrow{C}\theta_v$ and $\overrightarrow{D}\theta_v$ are subsequences of $\overrightarrow{\perp}$. But according to Definition 58, $\overrightarrow{C_S}$ is a subsequence of $\overrightarrow{C}$ and $\overrightarrow{D_S}$ is a subsequence of $\overrightarrow{D}$ and therefore $\overrightarrow{C_S}\theta_v \sqsubseteq \overrightarrow{C}\theta_v$ and $\overrightarrow{D_S}\theta_v \sqsubseteq \overrightarrow{D}\theta_v$. Then according to Lemma 2, $\overrightarrow{C_S}\theta_v$ and $\overrightarrow{D_S}\theta_v$ are subsequences of $\overrightarrow{\perp}$. Thus, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ are in $\overrightarrow{\mathcal{L}}_\perp$. □

Lemma 4 *Let $\overrightarrow{D}$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\perp$ and $\overrightarrow{C}$ is derived from $\overrightarrow{D}$ by removing some literals without changing the order of the remaining literals. Then, $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\perp$.*

Proof. $\overrightarrow{C}$ is derived from $\overrightarrow{D}$ by removing some literals and therefore $\overrightarrow{C}$ is a subsequence of $\overrightarrow{D}$. Let ε be the empty substitution, then we have $\overrightarrow{C}\varepsilon$ is a subsequence of $\overrightarrow{D}$ and $\varepsilon \subseteq \theta_\perp$ and according to Definition 55, $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$. □

Theorem 4 (Existence of $mgs_\perp$ in $\overrightarrow{\mathcal{L}}_\perp$) *For every ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$, there exists an $mgs_\perp$ of $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$.*

Proof. Let $\overrightarrow{\perp}$, θ_v and $\overrightarrow{\mathcal{L}}_\perp$ be as defined in Definition 53, $\theta_\perp$ be as defined in Definition 54 and $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$. Then $\overrightarrow{C}\theta_v$ and $\overrightarrow{D}\theta_v$ are subsequences of $\overrightarrow{\perp}$. Let S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ be as defined in Definition 58 such that S is a sequence of all pairs of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\perp$. According to Definition 53, $\overrightarrow{C_S}\theta_v = \overrightarrow{D_S}\theta_v$ and therefore $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ are unifiable and θ_v is a unifier for them. Let $\sigma \subseteq \theta_\perp$ be an mgu for $\{\overrightarrow{C_S}, \overrightarrow{D_S}\}$, n be the number of literals in $\overrightarrow{\perp}$ and $\overrightarrow{E}$ be defined as follows:

$$\overrightarrow{E} \quad = \quad (\bigvee_{i=1}^{n} l_i \text{ where } l_i \text{ is in } \overrightarrow{C} \text{ or in } \overrightarrow{D} \text{ and } l_i\theta_v \text{ is the i-th literal of } \overrightarrow{\perp})\sigma$$

We assume that the above disjunction notion with indexes from $i = 1$ to n means that the literals l_i of $\overrightarrow{E}$ follow the same order as literals in $\overrightarrow{\perp}$. We show that $\overrightarrow{E}$ is in $\overrightarrow{\mathcal{L}}_\perp$ and it is a $mgs_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$. Let h_1 and h_2 be the heads of $\overrightarrow{C}$ and $\overrightarrow{D}$ respectively. h_1 and h_2 are among compatible pair of literals in S and they are unified by σ and $\overrightarrow{E}$ has one literal in the head and therefore it is a definite ordered clause. Moreover, by definition $\overrightarrow{E}\theta_v$ is derived from $\overrightarrow{\perp}$ by removing some literals without changing the order of the remaining literals. Then $\overrightarrow{E}\theta_v$ is a subsequence of $\overrightarrow{\bot}$ and therefore $\overrightarrow{E}$ is in $\overrightarrow{\mathcal{L}}_\bot$. Now we show that $\overrightarrow{E}$ is a $mgs_\bot$ for $\overrightarrow{C}$ and $\overrightarrow{D}$. We have $\overrightarrow{C} \succeq_\bot \overrightarrow{E}$ and $\overrightarrow{D} \succeq_\bot \overrightarrow{E}$, since by definition $\overrightarrow{C}\sigma \sqsubseteq \overrightarrow{E}$ and $\overrightarrow{D}\sigma \sqsubseteq \overrightarrow{E}$ and $\sigma \subseteq \theta_\bot$. Suppose $\overrightarrow{F}$ is a clause in $\overrightarrow{\mathcal{L}}_\bot$ such that $\overrightarrow{C} \succeq_\bot \overrightarrow{F}$ and $\overrightarrow{D} \succeq_\bot \overrightarrow{F}$. In order to establish that $\overrightarrow{E}$ is an $mgs_\bot$ of $\overrightarrow{C}$ and $\overrightarrow{D}$, we need to prove $\overrightarrow{E} \succeq_\bot \overrightarrow{F}$. Let $\theta'_1, \theta'_2 \subseteq \theta_\bot$ be variable substitutions such that $\overrightarrow{C}\theta'_1 \sqsubseteq \overrightarrow{F}$ and $\overrightarrow{D}\theta'_2 \sqsubseteq \overrightarrow{F}$, h' be the head of $\overrightarrow{F}$ and $\theta' = \{\theta'_1 \cup \theta'_2\}$. Then, $h_1\theta' = h_1\theta'_1 = h'$ and $h_2\theta' = h_2\theta'_2 = h'$, so θ' is a unifier for h_1 and h_2. But σ is an mgu for h_1 and h_2 and so there is a substitution $\gamma \subseteq \theta_\bot$ such that $\theta' = \sigma\gamma$. According to the definition of $\overrightarrow{E}$, for every literal $l_i\sigma$ in $\overrightarrow{E}$, l_i is either in $\overrightarrow{C}$ or in $\overrightarrow{D}$. But $\overrightarrow{C}\theta' \sqsubseteq \overrightarrow{F}$ and $\overrightarrow{D}\theta' \sqsubseteq \overrightarrow{F}$ and therefore each literal $(l_i\sigma)\gamma$ in $\overrightarrow{E}\gamma$ is mapped to an equivalent literal $l_i\theta'$ in $\overrightarrow{F}$. Then $\overrightarrow{E}\gamma$ is a subsequence of $\overrightarrow{F}$ and $\gamma \subseteq \theta_\bot$ and we have $\overrightarrow{E} \succeq_\bot \overrightarrow{F}$. $\overrightarrow{E}$ is therefore an $mgs_\bot$ for $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\bot$. □

Example 20 *Let $\overrightarrow{C}$, $\overrightarrow{D}$, $\overrightarrow{\bot}$ and $\overrightarrow{\bot}_v$ be ordered clauses as defined below:*

$$\begin{aligned}
\overrightarrow{C} &= p(V_1, V_2) \leftarrow q(V_1, V_1), q(V_2, V_6) \\
\overrightarrow{D} &= p(V_1, V_2) \leftarrow q(V_3, V_1), q(V_2, V_2), r(V_1, V_2) \\
\overrightarrow{\bot} &= p(X, Y) \leftarrow q(X, X), q(Y, Y), r(X, Y), s(X, Y) \\
\overrightarrow{\bot}_v &= p(V_1, V_2) \leftarrow q(V_3, V_4), q(V_5, V_6), r(V_7, V_8), s(V_9, V_{10})
\end{aligned}$$

According to Definition 53, $\overrightarrow{C}$ and $\overrightarrow{D}$ are in $\overrightarrow{\mathcal{L}}_\bot$ as $\overrightarrow{C}\theta_v \sqsubseteq \overrightarrow{\bot}$ and $\overrightarrow{D}\theta_v \sqsubseteq \overrightarrow{\bot}$ where $\theta_v = \{V_1/X, V_2/Y, V_3/X, V_4/X, V_5/X, V_6/Y, V_7/X, V_8/Y, V_9/X, V_{10}/Y\}$. Then according to Definition 58, S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ are defined as follows:

$$\begin{aligned}
S &= (p(V_1, V_2), p(V_1, V_2)), (\neg q(V_1, V_1), \neg q(V_3, V_1)), \\
&\quad (\neg q(V_2, V_6), \neg q(V_1, V_2)) \\
\overrightarrow{C_S} &= p(V_1, V_2) \leftarrow q(V_1, V_1), q(V_2, V_6) \\
\overrightarrow{D_S} &= p(V_1, V_2) \leftarrow q(V_3, V_1), q(V_2, V_2)
\end{aligned}$$

Let $\sigma = \{X_3/V_1, V_6/V_2\}$ be an mgu for $\{\overrightarrow{C_S}, \overrightarrow{D_S}\}$, then $\sigma \subseteq \theta_\bot$ where

$$\begin{aligned}\theta_\perp \;=\;& \{V_3/V_1, V_4/V_1, V_5/V_1, V_7/V_1, V_9/V_1, V_4/V_3, V_5/V_3, V_7/V_3, V_9/V_3, V_5/V_4, V_7/V_4,\\ & V_9/V_4, V_7/V_5, V_9/V_5, V_9/V_7, V_6/V_2, V_8/V_2, V_{10}/V_2, V_8/V_6, V_{10}/V_6, V_{10}/V_8\}\end{aligned}$$

and according to Theorem 4, $mgs_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$ is defined as follows:

$$\begin{aligned} mgs_\perp(\overrightarrow{C}, \overrightarrow{D}) &= (\bigvee_{i=1}^{5} l_i \textit{ where } l_i \textit{ is in } \overrightarrow{C} \textit{ or in } \overrightarrow{D} \textit{ and } l_i\theta_v \textit{ is the i-th literal of } \overrightarrow{\perp})\sigma \\ &= (p(V_1, V_2) \leftarrow q(V_1, V_1), q(V_2, V_6), r(V_1, V_2))\sigma \\ &= p(V_1, V_2) \leftarrow q(V_1, V_1), q(V_2, V_2), r(V_1, V_2) \end{aligned}$$

◇

In the following we prove the existence of $lgg_\perp$ in $\overrightarrow{\mathcal{L}}_\perp$. We use the lgg for atoms as a bridge to define $lgg_\perp$ for clauses in $\overrightarrow{\mathcal{L}}_\perp$. This is similar to the proof for the existence of lgg for conventional clauses [NCdW97] adapted for ordered clauses and subsumption relative to a bottom clause. In the following we also use the same notion used in [NCdW97] for atomic generalisation and atomic representation of ordered clauses.

Definition 59 (Compatible clauses) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$ be ordered clauses. If $n = m$ and for every $i = 1, \ldots, n$, L_i and M_i have the same sign and predicate symbol, we say C and D are compatible clauses.*

Definition 60 (Atomic representation) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$ be an ordered clause. The atomic representation of clause $\overrightarrow{C}$ is denoted by $a(\overrightarrow{C}) = \vee(L_1, L_2, \ldots, L_n)$ where $\vee$ acts as a n-ary predicate symbol and $L_1, L_2, \ldots, L_n$ as terms.*

In this definition we assume an appropriate mapping between predicate symbols in $\overrightarrow{C}$ and function symbols in $a(\overrightarrow{C})$.

Definition 61 ($algg_\perp$) *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be two compatible ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ and $\overrightarrow{E}$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\perp$. $\overrightarrow{E}$ is called an atomic lgg relative to $\perp$, denoted by*

$\overrightarrow{E} = algg_\perp(\overrightarrow{C}, \overrightarrow{D})$ if $a(\overrightarrow{E}) = algg(a(\overrightarrow{C}), a(\overrightarrow{D}))$ where algg is lgg for atoms and $a(\overrightarrow{E})$, $a(\overrightarrow{C})$ and $a(\overrightarrow{D})$ are atomic representations of $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ as defined in Definition 60.

The following theorem proves the existence of $lgg_\perp$ in $\overrightarrow{\mathcal{L}}_\perp$. The proof is similar to the proof for the existence of lgg for conventional clauses (e.g. Theorem 14.27 in [NCdW97]) adapted for subsumption relative to a bottom clause.

Theorem 5 (Existence of $lgg_\perp$ in $\overrightarrow{\mathcal{L}}_\perp$) *For every ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$, there exists an $lgg_\perp$ of $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$.*

Proof. Let S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ be as defined in Definition 58 such that S is a sequence of all pairs of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\perp$. $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ are compatible clauses in $\overrightarrow{\mathcal{L}}_\perp$. Let $\overrightarrow{E} = algg_\perp(\overrightarrow{C_S}, \overrightarrow{D_S})$ where $algg_\perp$ is defined as in Definition 61. $\overrightarrow{E}$ is in $\overrightarrow{\mathcal{L}}_\perp$ and we show that it is a $lgg_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$. According to definition of $algg_\perp$ we have $\overrightarrow{E} \succeq_\perp \overrightarrow{C_S}$ and $\overrightarrow{E} \succeq_\perp \overrightarrow{D_S}$ and according to Definition 58, $\overrightarrow{C_S} \succeq_\perp \overrightarrow{C}$ and $\overrightarrow{D_S} \succeq_\perp \overrightarrow{D}$. By transitivity of $\succeq_\perp$ we have $\overrightarrow{E} \succeq_\perp \overrightarrow{C}$ and $\overrightarrow{E} \succeq_\perp \overrightarrow{D}$. Let $\overrightarrow{F} = N_1 \vee N_2 \vee \cdots \vee N_m$ be a clause in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{F} \succeq_\perp \overrightarrow{C}$ and $\overrightarrow{F} \succeq_\perp \overrightarrow{D}$. In order to establish that $\overrightarrow{E}$ is an $lgg_\perp$ of $\overrightarrow{C}$ and $\overrightarrow{D}$, we need to prove $\overrightarrow{F} \succeq_\perp \overrightarrow{E}$. Since $\overrightarrow{F} \succeq_\perp \overrightarrow{C}$ and $\overrightarrow{F} \succeq_\perp \overrightarrow{D}$, there are variable substitutions $\theta'_1, \theta'_2 \subseteq \theta_\perp$ and literals $L_1 \vee \cdots \vee L_m \sqsubseteq \overrightarrow{C}$ and $M_1 \vee \cdots \vee M_m \sqsubseteq \overrightarrow{D}$, such that $N_i\theta'_1 = L_i$ and $N_i\theta'_2 = M_i$, for every $1 \leq i \leq m$. Then $S' = (L_1, M_1), \ldots, (L_m, M_m)$ is a sequence of pairs of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\perp$ and $\overrightarrow{C_{S'}} = L_1 \vee L_2 \vee \cdots \vee L_m$ and $\overrightarrow{D_{S'}} = M_1 \vee M_2 \vee \cdots \vee M_m$ as defined in Definition 58. Let $\overrightarrow{G} = K_1 \vee K_2 \vee \cdots \vee K_m$ be an $algg_\perp(\overrightarrow{C_{S'}}, \overrightarrow{D_{S'}})$, where $algg_\perp$ is defined as in Definition 61 and there are substitutions $\sigma_1, \sigma_2 \subseteq \theta_\perp$ be such that $\overrightarrow{G}\sigma_1 = C_{S'}$ and $\overrightarrow{G}\sigma_2 = D_{S'}$. Since $(N_1 \vee N_2 \vee \cdots \vee N_m)\theta'_1 = C_{S'}$ and $(N_1 \vee N_2 \vee \cdots \vee N_m)\theta'_2 = D_{S'}$, we have $\overrightarrow{F} \succeq_\perp \overrightarrow{C_{S'}}$ and $\overrightarrow{F} \succeq_\perp \overrightarrow{D_{S'}}$. But $\overrightarrow{G} = algg_\perp(\overrightarrow{C_{S'}}, \overrightarrow{D_{S'}})$ and therefore $\overrightarrow{F} \succeq_\perp \overrightarrow{G}$. Moreover, S' is a subsequence of S and therefore $algg_\perp(\overrightarrow{C_{S'}}, \overrightarrow{D_{S'}})$ is a subsequence of $algg_\perp(\overrightarrow{C_S}, \overrightarrow{D_S})$ and according to Lemma 4 we have $\overrightarrow{G} \succeq_\perp \overrightarrow{E}$. Hence, $\overrightarrow{F} \succeq_\perp \overrightarrow{E}$ by transitivity of $\succeq_\perp$. □

According to Theorem 5, the $lgg_\perp$ of any pair of ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$ exists and can be computed by the following equation

$$lgg_\perp(\overrightarrow{C},\overrightarrow{D}) \quad = \quad algg_\perp(\overrightarrow{C_S},\overrightarrow{D_S})$$

where $algg_\perp$ is defined as in Definition 61 and S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ be as defined in Definition 58 such that S is a sequence of all pairs of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\perp$. Note that $\overrightarrow{C}$ and $\overrightarrow{D}$ are definite ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ with the same predicate symbol in the head and there is at least one pair of compatible literals from $\overrightarrow{C}$ and $\overrightarrow{D}$ relative to $\perp$ which is the pair of heads of $\overrightarrow{C}$ and $\overrightarrow{D}$. Hence, $lgg_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$ has at least one literal. Moreover, each literal from $\overrightarrow{C}$ and $\overrightarrow{D}$ can only be mapped to one literal from $\overrightarrow{\perp}$ and therefore each literal in $\overrightarrow{C}$ can be mapped to at most one literal from $\overrightarrow{D}$. Thus, $\overrightarrow{C}$ and $\overrightarrow{D}$ can have at most $min(|\overrightarrow{C}|,|\overrightarrow{D}|)$ pairs of compatible literals with respect to $\perp$ and accordingly $lgg_\perp(\overrightarrow{C},\overrightarrow{D})$ has at most $min(|\overrightarrow{C}|,|\overrightarrow{D}|)$ literals. Note that the lgg of $\overrightarrow{C}$ and $\overrightarrow{D}$ with respect to the general subsumption order has at most $|\overrightarrow{C}|\times|\overrightarrow{D}|$ literals as $\overrightarrow{C}$ and $\overrightarrow{D}$ can have at most $|\overrightarrow{C}|\times|\overrightarrow{D}|$ pairs of compatible literals.

Example 21 *Let $\overrightarrow{C}$, $\overrightarrow{D}$, $\overrightarrow{\perp}$, S, $\overrightarrow{C_S}$ and $\overrightarrow{D_S}$ be as defined in Example 20:*

$$\begin{aligned}
\overrightarrow{C} &= p(V_1,V_2) \leftarrow q(V_1,V_1), q(V_2,V_6)\\
\overrightarrow{D} &= p(V_1,V_2) \leftarrow q(V_3,V_1), q(V_2,V_2), r(V_1,V_2)\\
\overrightarrow{\perp} &= p(X,Y) \leftarrow q(X,X), q(Y,Y), r(X,Y), s(X,Y)\\
S &= (p(V_1,V_2),p(V_1,V_2)),(\neg q(V_1,V_1),\neg q(V_3,V_1)),\\
&\quad (\neg q(V_2,V_6),\neg q(V_1,V_2))\\
\overrightarrow{C_S} &= p(V_1,V_2) \leftarrow q(V_1,V_1), q(V_2,V_6)\\
\overrightarrow{D_S} &= p(V_1,V_2) \leftarrow q(V_3,V_1), q(V_2,V_2)
\end{aligned}$$

Then according to Theorem 5, $lgg_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$ is defined as follows:

$$lgg_\perp(\overrightarrow{C},\overrightarrow{D}) \quad = \quad \overrightarrow{E} = algg_\perp(\overrightarrow{C_S},\overrightarrow{D_S})$$

According to Definition 61 we have

$$\begin{aligned}
a(\overrightarrow{E}) &= algg(a(\overrightarrow{C_S}), a(\overrightarrow{D_S})) \\
&= algg(\vee(p(V_1, V_2), \neg q(V_1, V_1), \neg q(V_2, V_6)), \\
&\quad \vee(p(V_1, V_2), \neg q(V_3, V_1), \neg q(V_2, V_2))) \\
&= \vee(p(V_1, V_2), \neg q(V_3, V_1), \neg q(V_2, V_6)))
\end{aligned}$$

Then we have

$$lgg_\perp(\overrightarrow{C}, \overrightarrow{D}) = \overrightarrow{E} = p(V_1, V_2) \leftarrow q(V_3, V_1), q(V_2, V_6)$$

◇

Since we have proved the existence of $mgs_\perp$ and $lgg_\perp$ of any pair of ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$, it follows that $\overrightarrow{\mathcal{L}}_\perp$ ordered by subsumption relative to a bottom clause has a lattice structure.

Theorem 6 $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ *is a lattice.*

Proof. According to Theorem 3 $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ is a quasi-order. According to Theorems 4 and 5 each pair of ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ have a most general specialisation ($mgs_\perp$) and a least general generalisation ($lgg_\perp$) in $\overrightarrow{\mathcal{L}}_\perp$. Hence, $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ is a lattice. □

In the following we describe two different type of downward covers for the lattice $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and then we show that given ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C} \succ_\perp \overrightarrow{D}$, there is a finite chain of downward covers from $\overrightarrow{C}$ to a variant of $\overrightarrow{D}$.

Lemma 5 (Downward cover Type 1) *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{D} = \overrightarrow{C}\{y/x\}$ where x and y are distinct variables in $\overrightarrow{C}$. Then $\overrightarrow{D}$ is a downward cover of $\overrightarrow{C}$.*

Proof. Let $\overrightarrow{D} = \overrightarrow{C}\{y/x\}$ then $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ and $\overrightarrow{C}$ and $\overrightarrow{D}$ are not variant, hence $\overrightarrow{C} \succ_\perp \overrightarrow{D}$. Suppose there is a $\overrightarrow{E}$ in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C} \succ_\perp \overrightarrow{E} \succ_\perp \overrightarrow{D}$. Then there are variable substitutions $\theta, \sigma \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{E}$ and $\overrightarrow{E}\sigma \sqsubseteq \overrightarrow{D}$. We have $\overrightarrow{D} = \overrightarrow{C}\{y/x\}$, therefore

$\overrightarrow{C}$ and $\overrightarrow{D}$ only differ in variables y and x and have the same predicate symbols at the same positions and hence $\overrightarrow{E}$ has the same predicate symbols at the same positions as $\overrightarrow{C}$ and $\overrightarrow{D}$. If θ does not unify any variables in $\overrightarrow{C}$ then $\overrightarrow{C}$ and $\overrightarrow{E}$ would be variants, contradicting $\overrightarrow{C} \succ_\perp \overrightarrow{E}$. If θ unifies any other variable than x and y, then this contradicts $\overrightarrow{C}\theta\sigma \sqsubseteq \overrightarrow{D}$. Hence, θ must unify x and y and cannot unify any other variables. But then $\overrightarrow{C}\theta$ and $\overrightarrow{D}$ would be variants, contradicting $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{E} \succ_\perp \overrightarrow{D}$. Therefore such a $\overrightarrow{E}$ does not exist and $\overrightarrow{D}$ is a downward cover of $\overrightarrow{C}$. □

Lemma 6 (Downward cover Type 2) *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C}$ and $\overrightarrow{D}$ differ only in literal l in $\overrightarrow{D}$ and all variables in l are distinct new variables. Then $\overrightarrow{D}$ is a downward cover of $\overrightarrow{C}$.*

Proof. Let $\overrightarrow{C}$ and $\overrightarrow{D}$ differ only in literal l in $\overrightarrow{D}$ such that $l = p(x_1, x_2, \ldots, x_n)$ and all $x_1, x_2, \ldots, x_n$ are distinct new variables. Then $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ and $\overrightarrow{C}$ and $\overrightarrow{D}$ are not variant, hence $\overrightarrow{C} \succ_\perp \overrightarrow{D}$. Suppose there is a $\overrightarrow{E}$ in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C} \succ_\perp \overrightarrow{E} \succ_\perp \overrightarrow{D}$. Then there are variable substitutions $\theta, \sigma \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{E}$ and $\overrightarrow{E}\sigma \sqsubseteq \overrightarrow{D}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ differ only in literal l then there is a literal l' in $\overrightarrow{E}$ and $\overrightarrow{C}$ and $\overrightarrow{E}$ differ in l' because otherwise $\overrightarrow{C}$ and $\overrightarrow{E}$ are variants, contradicting $\overrightarrow{C} \succ_\perp \overrightarrow{E}$. Let $l' = p(x'_1, x'_2, \ldots, x'_n)$ then we have $p(x'_1, x'_2, \ldots, x'_n)\sigma = p(x_1, x_2, \ldots, x_n)$. However, if σ does not unify any of variables $x'_1, x'_2, \ldots, x'_n$ then $\overrightarrow{E}\theta$ and $\overrightarrow{D}$ would be variants, contradicting $\overrightarrow{E}\sigma \sqsubseteq \overrightarrow{D}$. If σ unifies any of variables $x'_1, x'_2, \ldots, x'_n$ then this contradicts the assumption that $x_1, x_2, \ldots, x_n$ are distinct variables. Therefore such a $\overrightarrow{E}$ does not exist and $\overrightarrow{D}$ is a downward cover of $\overrightarrow{C}$. □

Theorem 7 *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C} \succ_\perp \overrightarrow{D}$. Then there is a finite chain of downward covers from $\overrightarrow{C}$ to a variant of $\overrightarrow{D}$.*

Proof. Let $\overrightarrow{C} \succ_\perp \overrightarrow{D}$ then there is a variable substitution $\theta \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. Let $\overrightarrow{C}$ and $\overrightarrow{D}$ differ in literals $l_1, l_2, \ldots, l_j$ in $\overrightarrow{D}$, $j \geq 0$. For every l_i there is a most general literal l'_i of which l_i is an instance. Assume that for every $1 \leq i \leq j-1$, the variables in l'_{i+1} do not appear in $\overrightarrow{C}$ or in literals $l_1, \ldots, l_i$. Then we have a finite chain of downward Type 2 covers (Lemma 6) of length

j, $\overrightarrow{C} = \overrightarrow{C}_0 \succ_\perp \overrightarrow{C}_1 \succ_\perp \cdots \succ_\perp \overrightarrow{C}_j$ where $\overrightarrow{C}_i$ and $\overrightarrow{C}_{i+1}$ differ only in literal l'_{i+1}. Moreover, there is a substitution θ' such that $\overrightarrow{C}_j\theta' = \overrightarrow{D}$ where $\theta \subseteq \theta' \subseteq \theta_\perp$. Suppose that the cardinality of θ' is k, i.e. $\theta' = \{y_1/x_1, y_2/x_2, \ldots, y_k/x_k\}$. Then we have $\theta' = \theta'_1\theta'_2 \ldots \theta'_k = \{y_1/x_1\}\{y_2/x_2\} \ldots \{y_k/x_k\}$. Hence, there is a finite chain of downward Type 1 covers (Lemma 5) of length k from $\overrightarrow{C}_j$ to a variant of $\overrightarrow{D}$ and therefore there exists a finite chain of downward covers of length $j+k$ from $\overrightarrow{C}$ to a variant of $\overrightarrow{D}$ □

4.2 Morphism between the lattice of bounded subsumption and an atomic lattice

In the following, we discuss a morphism between the lattice $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and an atomic lattice. First we define the set of atoms which are generalisations of the atomic representation of $\overrightarrow{\perp}$.

Definition 62 ($\mathcal{A}_\perp$) *Let $\overrightarrow{\perp}$ and θ_v be as defined in Definition 53, $a(\overrightarrow{\perp}) = \vee(t_1, t_2, \ldots, t_n)$ be the atomic representation of $\overrightarrow{\perp}$ as defined in Definition 60, $A_v = \vee(w_1, w_2, \ldots, w_n)$ be $a(\overrightarrow{\perp})$ with all term positions t_j populated with new and distinct variables w_j. Let σ_v be a substitution such that $A_v\sigma_v = a(\overrightarrow{\perp})$. Atom A is in $\mathcal{A}_\perp$ if $A\theta_v\sigma_v = a(\overrightarrow{\perp})$.*

Example 22 *Let $a(\overrightarrow{\perp})$, A_v and atoms A_1, A_2, A_3, A_4 be defined as follows:*

$$\begin{array}{rccccc}
A_1 = & \vee(p(V_1), & W_2, & \neg r(V_3), & W_4, & \neg s(V_6, V_7)) \\
A_2 = & \vee(p(V_1), & W_2, & \neg r(V_1), & W_4, & \neg s(V_6, V_1)) \\
A_3 = & \vee(p(V_1), & W_2, & \neg r(V_1), & \neg s(V_4, V_5), & W_5) \\
A_4 = & \vee(p(V_1), & \neg q(V_1), & \neg r(V_1), & W_4, & \neg s(V_6, V_1)) \\
A_v = & \vee(W_1, & W_2, & W_3, & W_4, & W_5) \\
a(\overrightarrow{\perp}) = & \vee(p(X), & \neg q(X), & \neg r(X), & \neg s(X,Y), & \neg s(Y,X))
\end{array}$$

Then according to Definition 62, A_1, A_2, A_3, A_4 are in $\mathcal{A}_\perp$ as $A_1\theta_v\sigma_v = A_2\theta_v\sigma_v = A_3\theta_v\sigma_v = A_4\theta_v\sigma_v = a(\overrightarrow{\perp})$, where $\theta_v = \{V_1/X, V_2/X, V_3/X, V_4/X, V_5/Y, V_6/Y, V_7/X\}$ and $\sigma_v = \{W_1/p(X), W_2/\neg q(X), W_3/\neg r(X), W_4/\neg s(X,Y), W_5/\neg s(Y,X)\}$.

◇

Definition 63 (Mapping function cl) *Let $\overrightarrow{\mathcal{L}}_\perp$ and $\mathcal{A}_\perp$ be as defined in Definition 53 and Definition 62 and A be an atom in $\mathcal{A}_\perp$. The mapping function $cl : \mathcal{A}_\perp \rightarrow$*

$\overrightarrow{\mathcal{L}}_\perp$ *is defined as follows:*

$$cl(A) = (\bigvee_{i=1}^{n} l_i \text{ where } l_i \text{ is the i-th term from } A \text{ which is not a variable})$$

As in Definition 60 we assume an appropriate mapping between predicate symbols in $cl(A)$ and function symbols in A. Note that in Definition 63, non-variable terms in A represent literals in $cl(A)$ and variables in A represent the absence of literals from $\overrightarrow{\perp}$ in $cl(A)$. These variables, which correspond to variables w_i in σ_v, are always distinct variables because, according to Definition 62, they can only be substituted by distinct non-variable terms from $a(\overrightarrow{\perp})$. Hence, in Definition 63, if l_i is a variable then it is always distinct and cannot be unified with other variables in A. In order to simplify the representation we replace all such variables (i.e. w_i) by the symbol '_'.

Example 23 *Let $\overrightarrow{\perp}$, $\overrightarrow{C}$, $\overrightarrow{D}$, $\overrightarrow{E}$ and $\overrightarrow{F}$ be defined as follows:*

$$\begin{aligned}
\overrightarrow{C} &= p(V_1) \leftarrow r(V_3), s(V_6, V_7) \\
\overrightarrow{D} &= p(V_1) \leftarrow r(V_1), s(V_6, V_1) \\
\overrightarrow{E} &= p(V_1) \leftarrow r(V_1), s(V_4, V_5) \\
\overrightarrow{F} &= p(V_1) \leftarrow q(V_1), r(V_1), s(V_6, V_1) \\
\overrightarrow{\perp} &= p(X) \leftarrow q(X), r(X), s(X, Y), s(Y, X)
\end{aligned}$$

and $a(\overrightarrow{\perp})$ and atoms A_1, A_2, A_3 and A_4 in $\mathcal{A}_\perp$ be defined as follows:

$$\begin{array}{rlllll}
A_1 = & \vee(p(V_1), & \_, & \neg r(V_3), & \_, & \neg s(V_6, V_7)) \\
A_2 = & \vee(p(V_1), & \_, & \neg r(V_1), & \_, & \neg s(V_6, V_1)) \\
A_3 = & \vee(p(V_1), & \_, & \neg r(V_1), & \neg s(V_4, V_5), & \_) \\
A_4 = & \vee(p(V_1), & \neg q(V_1), & \neg r(V_1), & \_, & \neg s(V_6, V_1)) \\
a(\overrightarrow{\perp}) = & \vee(p(X), & \neg q(X), & \neg r(X), & \neg s(X, Y), & \neg s(Y, X))
\end{array}$$

Then atoms A_1, A_2, A_3 and A_4 correspond to ordered clauses $\overrightarrow{\perp}$, $\overrightarrow{C}$, $\overrightarrow{D}$, $\overrightarrow{E}$ and we have $\overrightarrow{C} = cl(A_1)$, $\overrightarrow{D} = cl(A_2)$, $\overrightarrow{E} = cl(A_3)$, $\overrightarrow{F} = cl(A_4)$. $\diamond$

Theorem 8 *Let $\mathcal{A}_\perp$ be as defined in Definition 62 and the mapping function cl as defined in Definition 63. Let A and B be atoms in $\mathcal{A}_\perp$, then we have $cl(A) \succeq_\perp cl(B)$ if and only if $A \succeq B$.*

Proof. Let $A = \vee(L_1, \ldots, L_n)$, $B = \vee(M_1, \ldots, M_n)$ and $a(\overrightarrow{\bot}) = \vee(N_1, \ldots, N_n)$. Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses such that $\overrightarrow{C} = cl(A)$ and $\overrightarrow{D} = cl(B)$.

$\Rightarrow$: Suppose $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$, then there exists a substitution $\theta \subseteq \theta_\bot$ such that $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$ and therefore for each literal in $\overrightarrow{C}\theta$ there is an equivalent literal in $\overrightarrow{D}$. Hence, for each term L_i in A if L_i is non-variable term then it corresponds to a literal in $\overrightarrow{C}$ and therefore there exist a non-variable term M_i in B such that $L_i\theta = M_i$. If L_i is a variable then there exist a substitution θ' such that $L_i\theta' = M_i$ where $L_i/M_i \in \theta'$ and M_i is a variable or non-variable term in B. Hence, for each term L_i in A there is a substitution $\sigma = \theta \cup \theta'$ such that $L_i\sigma = M_i$. This implies $A\sigma = B$ and therefore $A \succeq B$.

$\Leftarrow$: Suppose $A \succeq B$, then there exists a substitution σ such that $A\sigma = B$. Then for each term L_i in A we have term M_i in B such that $L_i\sigma = M_i$. Moreover, A and B are in $\mathcal{A}_\bot$ and according to Definition 62 both terms L_i and M_i correspond to the same term N_i from $a(\overrightarrow{\bot})$ and we have $L_i\theta_v = M_i\theta_v = N_i$. If L_i is a non-variable term then according to Definition 62 M_i is also a non-variable term. Then according to Definition 54, there is a substitution $\theta \subseteq \theta_\bot$ such that for non-variable terms L_i and M_i we have $L_i\theta = M_i$. Hence, for each literal in $\overrightarrow{C}\theta$ there is an equivalent literal in $\overrightarrow{D}$. Then $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$ and $\theta \subseteq \theta_\bot$ and therefore we have $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$. □

Example 24 *Let $\overrightarrow{\bot}$ be the bottom clause and $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be the ordered clauses as in Figure 3.1. Atoms A_1, A_2 and A_3 in Figure 3.2 correspond to ordered clauses $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ and we have $\overrightarrow{C} = cl(A_1)$, $\overrightarrow{D} = cl(A_2)$ and $\overrightarrow{E} = cl(A_3)$. We have $\overrightarrow{C} \succeq_\bot \overrightarrow{D}$ and $A_1 \succeq A_2$ as shown in Figure 3.2.a and $\overrightarrow{C} \not\succeq_\bot \overrightarrow{E}$ and $A_1 \not\succeq A_3$ as shown in Figure 3.2.b.* ◇

Theorem 9 *The mapping function $cl : \mathcal{A}_\bot \to \overrightarrow{\mathcal{L}}_\bot$ as defined in Definition 63 is an order-isomorphism.*

Proof. First we show that the mapping function cl is onto, i.e. for each ordered clause $\overrightarrow{C}$ in $\overrightarrow{\mathcal{L}}_\bot$, there is $A \in \mathcal{A}_\bot$ such that $\overrightarrow{C} = cl(A)$. Let $\overrightarrow{\bot} = N_1 \vee N_2 \vee \cdots \vee N_n$ and $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_n$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\bot$, then we have $\overrightarrow{C}\theta_v \sqsubseteq \overrightarrow{\bot}$. Let $A = \vee(M_1, M_2, \ldots, M_n)$ be an atom in $\mathcal{A}_\bot$ such that the i-th term M_i is L_j if $L_j\theta_v$ is the i-th literal N_i in $\overrightarrow{\perp}$; otherwise M_i is a variable w_i such that $w_i/t_i \in \sigma_v$. Then by construction $\overrightarrow{C} = cl(A)$ and $A\theta_v\sigma_v = a(\overrightarrow{\perp})$. Hence, the mapping function cl is onto. Moreover, according to Theorem 8, the mapping function cl is an order-embedding. Then according to Definition 26, cl is an order-isomorphism. $\square$

We have shown that $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ is a lattice. It is also known that the atomic subsumption defines a lattice [Rey69]. The proposition below follows directly from Theorem 9 and Proposition 7.

Proposition 11 *The mapping function $cl : \mathcal{A}_\perp \rightarrow \overrightarrow{\mathcal{L}}_\perp$ as defined in Definition 63 is a lattice isomorphism and lattices $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{A}_\perp, \succeq \rangle$ are two isomorphic lattices.*

The proposition below follows from cl being a lattice isomorphism.

Proposition 12 *Let $\mathcal{A}_\perp$ and mapping function cl be defined as in Definition 63 and A and B be atoms in $\mathcal{A}_\perp$. The mapping function $cl : \mathcal{A}_\perp \rightarrow \overrightarrow{\mathcal{L}}_\perp$ is join-preserving and meet-preserving that is:*

1. $mgs_\perp(cl(A), cl(B)) = cl(mgs(A, B))$
2. $lgg_\perp(cl(A), cl(B)) = cl(lgg(A, B))$

Example 25 *Let $a(\overrightarrow{\perp})$ and atoms A and B in $\mathcal{A}_\perp$ be as defined below:*

$$\begin{aligned}
A &= \vee(p(V_1,V_2), \neg q(V_1,V_1), \neg q(V_2,V_6), \_, \_) \\
B &= \vee(p(V_1,V_2), \neg q(V_3,V_1), \neg q(V_2,V_2), \neg r(V_1,V_2), \_) \\
a(\overrightarrow{\perp}) &= \vee(p(X,Y), \neg q(X,X), \neg q(Y,Y), \neg r(X,Y), \neg s(X,Y))
\end{aligned}$$

According to Definition 63, $cl(A)$ and $cl(B)$ are defined as follows:

$$\begin{aligned}
\overrightarrow{C} = cl(A) &= p(V_1,V_2) \leftarrow q(V_1,V_1), q(V_2,V_6) \\
\overrightarrow{D} = cl(B) &= p(V_2,V_2) \leftarrow q(V_3,V_1), q(V_2,V_2), r(V_1,V_2)
\end{aligned}$$

$mgs(A,B)$ and $lgg(A,B)$ are defined as follows:

$$\begin{aligned} mgs(A,B) &= \vee(p(V_1,V_2),\neg q(V_1,V_1),\neg q(V_2,V_2),\neg r(V_1,V_2),\_) \\ lgg(A,B) &= \vee(p(V_1,V_2),\neg q(V_3,V_1),\neg q(V_2,V_6),\_,\_) \end{aligned}$$

Then according to Proposition 11, $mgs_\perp$ and $lgg_\perp$ for $\overrightarrow{C}$ and $\overrightarrow{D}$ are defined as follows:

$$\begin{aligned} mgs_\perp(\overrightarrow{C},\overrightarrow{D}) &= p(V_1,V_2) \leftarrow q(V_1,V_1), q(V_2,V_2), r(V_1,V_2) \\ lgg_\perp(\overrightarrow{C},\overrightarrow{D}) &= p(V_1,V_2) \leftarrow q(V_3,V_1), q(V_2,V_6) \end{aligned}$$

$\diamond$

As previously discussed in this chapter, for ordered clauses $\overrightarrow{C}$ and $\overrightarrow{D}$ in $\overrightarrow{\mathcal{L}}_\perp$ we have at most $min(|\overrightarrow{C}|,|\overrightarrow{D}|)$ pairs of compatible literals relative to $\perp$ and accordingly $lgg_\perp(\overrightarrow{C},\overrightarrow{D})$ has at most $min(|\overrightarrow{C}|,|\overrightarrow{D}|)$ literals. Similarly, Proposition 11 suggests that $lgg_\perp(\overrightarrow{C},\overrightarrow{D})$ has at most $|\overrightarrow{\perp}|$ literals as $|\overrightarrow{C}|$ and $|\overrightarrow{D}|$ are bounded by $|\overrightarrow{\perp}|$.

It is known that general subsumption testing is NP-complete [GJ79]. However, as shown in this section, subsumption testing relative to a bottom clause can be mapped to atomic subsumption testing. Atomic subsumption testing can be reduced to a unification problem which can be decided in linear time [GL85].

4.3 Morphism between the atomic lattice and the lattice of partitions

In this chapter, we have shown that the subsumption order relative to a bottom clause defines a lattice (i.e. $\langle\overrightarrow{\mathcal{L}}_\perp,\succeq_\perp\rangle$) and this lattice is isomorphic to an atomic lattice (i.e. $\langle\mathcal{A}_\perp,\succeq\rangle$). In this section we show that the lattice $\langle\overrightarrow{\mathcal{L}}_\perp,\succeq_\perp\rangle$ can also be mapped to a lattice of partitions. This mapping will be used in the next chapter for encoding the bounded subsumption lattice. In the following, we first show the morphism between the function free atomic lattice and the lattice of variable partitions.

Definition 64 *Let Π_m be the set of all partitions on $\{1,2,\ldots,m\}$ and π_1 and π_2 be partitions in Π_m. We say π_1 is finer than π_2, denoted by $\pi_1 \leq \pi_2$ if and only if for* *each block B_1 in π_1 there is a block B_2 in π_2 such that $B_1 \subseteq B_2$. π_1 is properly finer than π_2, denoted by $\pi_1 < \pi_2$, if $\pi_1 \leq \pi_2$ and $\pi_2 \not\leq \pi_1$.*

It is known [DP02] that Π_m is partially ordered by $\leq$ and that $\langle \Pi_m, \leq \rangle$ is a lattice.

Proposition 13 *Let Π_m and $\leq$ be as defined in Definition 64. Then $\langle \Pi_m, \leq \rangle$ is a lattice.*

Definition 65 (Mapping function π) *Let $\mathcal{A}_m$ be the set of all atoms in a language with only one m-ary predicate symbol and with no constant and function symbols. The mapping function $\pi : \mathcal{A}_m \to \Pi_m$ is defined to map any atom $A = p(v_1, v_2, \ldots, v_m)$ in $\mathcal{A}_m$ to a partition π in Π_m such that for each block B in π, $\{i, j\} \subseteq B$ if and only if variables v_i and v_j are the same.*

Example 26 *Let $A = p(X, Y, X, Z, Y, X)$ be an atom in $\mathcal{A}_6$. Then $\pi(A) = \{\{1, 3, 6\}, \{2, 5\}, \{4\}\}$.* $\diamond$

Lemma 7 *Let $\mathcal{A}_m$ be as defined in Definition 65 and $A_1 = p(u_1, .., u_m)$ and $A_2 = p(v_1, .., v_m)$ be atoms in $\mathcal{A}_m$. There exists a variable substitution θ such that $A_1\theta = A_2$ if and only if for any pair of variables u_i and u_j in A_1 if u_i and u_j are the same then variables v_i and v_j in A_2 are the same.*

Proof. $\Rightarrow$: Suppose that there exists a variable substitution θ such that $p(u_1, .., u_m)\theta = p(v_1, .., v_m)$. Let $\{u_i/v_i, u_j/v_j\} \subseteq \theta$. Then according to Definition 9, u_i and u_j must be distinct variables. Hence, if variables u_i and u_j are the same then variables v_i and v_j are the same.

$\Leftarrow$: Suppose that for any pair of variables u_i and u_j in A_1 if u_i and u_j are the same then variables v_i and v_j in A_2 are the same. Then a function can be defined which maps each variable u_i from A_1 to a variable v_i from A_2. Then according to Definition 9, there is a variable substitution θ such that $A_1\theta = A_2$. $\square$

Example 27 *Let A_1 and A_2 be atoms in $\mathcal{A}_6$ as defined below*

$$\begin{aligned} A_1 &= p(X_1, Y_1, X_1, Z_1, Y_1, X_1) \\ A_2 &= p(X_2, Y_2, X_2, X_2, Y_2, X_2) \end{aligned}$$

We have $A_1\theta = A_2$ where $\theta = \{X_1/X_2, Y_1/Y_2, Z_1/X_2\}$. For any pair of variables u_i and u_j in A_1 if u_i and u_j are the same then variables v_i and v_j in A_2 are the same. For example, u_1 and u_3 in A_1 represent the same variable (i.e. X_1) and v_1 and v_3 in A_2 represent the same variable (i.e. X_2). $\diamond$

Theorem 10 *Let $\mathcal{A}_m$ and mapping function π be as defined in Definition 65 and A_1 and A_2 be atoms in $\mathcal{A}_m$. $A_1 \succeq A_2$ if and only if $\pi(A_1) \leq \pi(A_2)$.*

Proof. $\Rightarrow$: Let $A_1 = p(u_1, .., u_m)$ and $A_2 = p(v_1, .., v_m)$ such that $A_1 \succeq A_2$. Then according to Definition 35, there exists a substitution θ such that $p(u_1, .., u_m)\theta = p(v_1, .., v_m)$. According to Lemma 7, for any pair of variables u_i and u_j representing the same variable in A_1, variables v_i and v_j represent the same variable in A_2. Then according to Definition 65, if $\{i, j\} \subseteq B_1$ where $B_1 \in \pi(A_1)$ then there is $\{i, j\} \subseteq B_2$ where $B_2 \in \pi(A_2)$. Then according to Definition 64, $\pi(A_1) \leq \pi(A_2)$.

$\Leftarrow$: Let $A_1 = p(u_1, .., u_m)$ and $A_2 = p(v_1, .., v_m)$ such that $\pi(A_1) \leq \pi(A_2)$. Then according to Definition 64, for each block B_1 in $\pi(A_1)$ there is a block B_2 in $\pi(A_2)$ such that $B_1 \subseteq B_2$. Hence, for each $\{i, j\} \subseteq B_1$ where $B_1 \in \pi(A_1)$, there is $\{i, j\} \subseteq B_2$ where $B_2 \in \pi(A_2)$. Then according to Definition 65, for any pair of variables u_i and u_j representing the same variable in A_1, variables v_i and v_j represent the same variable in A_2. According to Lemma 7, there exists a variable substitution θ such that $p(u_1, .., u_m)\theta = p(v_1, .., v_m)$ and therefore $A_1 \succeq A_2$. $\square$

Theorem 11 *The mapping function $\pi : \mathcal{A}_m \rightarrow \Pi_m$ as defined in Definition 65 is an order-isomorphism.*

Proof. First we show that the mapping function π is onto. Let π be a partition in Π_m. We show that there is an atom A in $\mathcal{A}_m$ such that $\pi(A) = \pi$. Let $A = p(v_1, v_2, \ldots, v_m)$ be an atom in $\mathcal{A}_m$ such that for each block B in π and for each $\{i, j\} \subseteq B$, variables v_i and v_j are the same. Then according to Definition 65 we have $\pi(A) = \pi$ and therefore π is onto. Moreover, according to Theorem 10 and Definition 26, the mapping function π is order-embedding. Then according to Definition 26, π is an order-isomorphism. $\square$

According to Proposition 13, $\langle \Pi_m, \leq \rangle$ is a lattice. The proposition below follows directly from Theorem 11 and Proposition 7.

Proposition 14 *The mapping function $\pi : \mathcal{A}_m \rightarrow \Pi_m$ as defined in Definition 65 is a lattice isomorphism and lattices $\langle \Pi_m, \leq \rangle$ and $\langle \mathcal{A}_m, \succeq \rangle$ are two isomorphic lattices.*

Proposition 14 shows the morphism between a function free atomic lattice and the lattice of variable partitions. In the following we study the mapping of $\mathcal{A}_\perp$ to variable partitions relative to a bottom clause. In the following we first define the set of valid variable positions and variable partitions relative to a bottom clause. The set of valid variable positions relative to a bottom clause is a set of variable positions in $a(\overrightarrow{\perp})$ which are valid with respect to a subset of terms in $a(\overrightarrow{\perp})$, i.e. if a variable position from a term is included then all variable positions from that term should be included.

Definition 66 (Variable positions and partitions relative to $\overrightarrow{\perp}$) *Let $\overrightarrow{\perp}$ and $a(\overrightarrow{\perp})$ be as defined in Definition 62 and $\mathcal{A}_m$ and mapping function π be as defined in Definition 65. Let $A_\perp = p(v_1, v_2, \ldots, v_m)$ be an atom in $\mathcal{A}_m$ such that $v_1, v_2, \ldots, v_m$ are variables in $a(\overrightarrow{\perp})$ despite function symbols. The set of valid variable positions relative to a bottom clause, denoted by $\mathcal{V}_\perp$, is defined as follows:*

$$\mathcal{V}_\perp = \{V \subseteq \{1, \ldots, m\} | i \in V \Rightarrow j \in V, \textit{ whenever } v_i, v_j \textit{ are in the same term in } a(\overrightarrow{\perp})\}$$

The set of valid variable partitions relative to a bottom clause, denoted by $\Pi_\perp$, is defined as follows:

$$\Pi_\perp = \{\pi | \pi \textit{ is a partition on } V \textit{ for some } V \textit{ in } \mathcal{V}_\perp \textit{ and } \pi \leq \pi(A_\perp)\}$$

Example 28 *Let $a(\overrightarrow{\perp})$ be defined as follows:*

$$a(\overrightarrow{\perp}) = \vee(p(\underset{v_1}{X}), \neg q(\underset{v_2}{X}), \neg r(\underset{v_3}{X}), \neg s(\underset{v_4}{X}, \underset{v_5}{Y}), \neg s(\underset{v_6}{Y}, \underset{v_7}{X}))$$

Then according to Definition 66, we have $\{1,3,6,7\} \in \mathcal{V}_\perp$ *and* $\{1,3,5,7\} \notin \mathcal{V}_\perp$*. We also have* $A_\perp = p(X,X,X,X,Y,Y,X)$ *and* $\pi(A_\perp) = \{\{1,2,3,4,7\},\{5,6\}\}$ *and therefore* $\{\{1,2,3,7\},\{6\}\} \in \Pi_\perp$ *and* $\{\{1,2,3,7,6\}\} \notin \Pi_\perp$*.*

The following definition is similar to Definition 65 but defined for $\mathcal{A}_\perp$ and $\Pi_\perp$ instead of $\mathcal{A}_m$ and Π_m.

Definition 67 (Mapping function $\pi_\perp$) *Let* $\mathcal{A}_\perp$ *and* $a(\overrightarrow{\perp})$ *be as defined in Definition 62 and* $\Pi_\perp$ *be as defined in Definition 66. The mapping function* $\pi_\perp : \mathcal{A}_\perp \rightarrow \Pi_\perp$ *is defined to map any atom* A *in* $\mathcal{A}_\perp$ *to a partition* π *in* $\Pi_\perp$ *such that for each block* B *in* π*,* $\{i,j\} \subseteq B$ *if and only if variables* v_i *and* v_j *in* A *are the same where* i *and* j *correspond to the* i*th and* j*th variable positions in* $a(\overrightarrow{\perp})$*.*

Example 29 *Let* $a(\overrightarrow{\perp})$ *and atoms* A_1 *,* A_2*,* A_3 *and* A_4 *in* $\mathcal{A}_\perp$ *be defined as follows:*

$$\begin{array}{rlllll}
A_1 = & \vee(p(V_1), & \_, & \neg r(V_3), & \_, & \neg s(V_6,V_7)) \\
A_2 = & \vee(p(V_1), & \_, & \neg r(V_1), & \_, & \neg s(V_6,V_1)) \\
A_3 = & \vee(p(V_1), & \_, & \neg r(V_1), & \neg s(V_4,V_5), & \_) \\
A_4 = & \vee(p(V_1), & \neg q(V_1), & \neg r(V_1), & \_, & \neg s(V_6,V_1)) \\
a(\overrightarrow{\perp}) = & \vee(p(X), & \neg q(X), & \neg r(X), & \neg s(X,Y), & \neg s(Y,X))
\end{array}$$

According to Definition 67, $\pi_\perp(A_1)$*,* $\pi_\perp(A_2)$*,* $\pi_\perp(A_3)$ *,* $\pi_\perp(A_4)$ *and* $\pi_\perp(a(\overrightarrow{\perp}))$ *are as follows:*

$$\begin{aligned}
\pi_1 &= \pi_\perp(A_1) = \{\{1\},\{3\},\{6\},\{7\}\} \\
\pi_2 &= \pi_\perp(A_2) = \{\{1,3,7\},\{6\}\} \\
\pi_3 &= \pi_\perp(A_3) = \{\{1,3\},\{4\},\{5\}\} \\
\pi_4 &= \pi_\perp(A_3) = \{\{1,2,3,7\},\{6\}\} \\
\pi_\perp &= \pi_\perp(a(\overrightarrow{\perp})) = \{\{1,2,3,4,7\},\{5,6\}\}
\end{aligned}$$

Note that $\pi_\perp = \pi_\perp(a(\overrightarrow{\perp})) = \pi(A_\perp)$ *and* π_1*,* π_2*,* π_3 *and* π_4 *are partitions in* $\Pi_\perp$ *and we have* $\pi_1 \leq \pi_2 \leq \pi_4 \leq \pi_\perp$ *and* $\pi_3 \leq \pi_\perp$*.*

The following lemma and theorem are similar to Lemma 7 and Theorem 10 for $\langle \Pi_m, \leq \rangle$ and $\langle \mathcal{A}_m, \succeq \rangle$ adapted for $\langle \Pi_\perp, \leq \rangle$ and $\langle \mathcal{A}_\perp, \succeq \rangle$.

Lemma 8 *Let $\mathcal{A}_\perp$ be as defined in Definition 62 and $A_1 = p(s_1, .., s_n)$ and $A_2 = p(t_1, .., t_n)$ be atoms in $\mathcal{A}_\perp$. There exists a substitution θ such that $A_1\theta = A_2$ if and only if the following two conditions hold: (1) for each s_k if s_k is a non-variable term then t_k is a non-variable term and (2) for any pair of variables u_i and u_j in A_1, where i and j correspond to the ith and jth variable positions in $a(\overrightarrow{\perp})$, if u_i and u_j are the same then variables v_i and v_j in A_2 are the same.*

Proof. $\Rightarrow$: Suppose that there exists a variable substitution θ such that $p(s_1, .., s_n)\theta = p(t_1, .., t_n)$. Then for each $s_k/t_k \in \theta$, if s_k is a non-variable terms in A_1 then t_k is a non-variable term in A_2. For each s_k in A_1 if s_k is a variable then it correspond to variable w_k in Definition 62 and therefore it is a distinct new variable and cannot be the same as any other variable in A_1. Let u_i and u_j be variables in some non-variable terms in A_1 and v_i and v_j be variables in some non-variable terms in A_2 where i and j correspond to the ith and jth variable positions in $a(\overrightarrow{\perp})$. Then $\{u_i/v_i, u_j/v_j\} \subseteq \theta$ and therefore according to Definition 9, u_i and u_j must be distinct variables. Hence, if variables u_i and u_j are the same then variables v_i and v_j are the same.

$\Leftarrow$: Suppose that for each s_k in A_1 if s_k is a non-variable term then t_k is a non-variable term in A_2 and that for any pair of variables u_i and u_j in A_1, where i and j correspond to the ith and jth variable positions in $a(\overrightarrow{\perp})$, if u_i and u_j are the same then variables v_i and v_j in A_2 are the same. Then a function can be defined which maps every variable u_i to a variable v_i. Hence, for each s_k, if s_k is a variable then it is mapped to a (variable or non-variable) term t_k and if s_k is a non-variable term then each variable in s_k is mapped to a variable in the non-variable term t_k. Hence, a function can be defined which maps each variable from A_1 to a corresponding variable or term from A_2. Then according to Definition 9, there is a substitution θ such that $A_1\theta = A_2$. $\square$

Theorem 12 *Let $\mathcal{A}_\perp$ and mapping function $\pi_\perp$ be as defined in Definition 67 and A_1 and A_2 be atoms in $\mathcal{A}_\perp$. $A_1 \succeq A_2$ if and only if $\pi_\perp(A_1) \leq \pi_\perp(A_2)$.*

Proof. $\Rightarrow$: Let $A_1 = p(s_1, .., s_n)$ and $A_2 = p(t_1, .., t_n)$ such that $A_1 \succeq A_2$. Then according to Definition 35, there exists a substitution θ such that $p(s_1, .., s_n)\theta = p(t_1, .., t_n)$. According to Lemma 8, for any pair of variables u_i and u_j in A_1, where i and j correspond to the ith and jth variable positions in $a(\overrightarrow{\perp})$, if u_i and u_j are the same then variables v_i and v_j in A_2 are the same. Then according to Definition 67, if $\{i, j\} \subseteq B_1$ where $B_1 \in \pi_\perp(A_1)$ then there is $\{i, j\} \subseteq B_2$ where $B_2 \in \pi_\perp(A_2)$. Then according to Definition 64, $\pi_\perp(A_1) \leq \pi_\perp(A_2)$.

$\Leftarrow$: Let $A_1 = p(s_1, .., s_n)$ and $A_2 = p(t_1, .., t_n)$ such that $\pi_\perp(A_1) \leq \pi_\perp(A_2)$. Then according to Definition 64 for each block B_1 in $\pi_\perp(A_1)$ there is a block B_2 in $\pi_\perp(A_2)$ such that $B_1 \subseteq B_2$. Hence, for each $\{i, j\} \subseteq B_1$ where $B_1 \in \pi(A_1)$, there is $\{i, j\} \subseteq B_2$ where $B_2 \in \pi(A_2)$. Then according to Definition 67, for any pair of variables u_i and u_j in A_1, if u_i and u_j are the same then variables v_i and v_j in A_2 are the same. Moreover, $\pi_\perp(A_1) \leq \pi_\perp(A_2)$ and according to Definition 64, if i is in some block B_1 in $\pi_\perp(A_1)$ then it is in some block B_2 in $\pi_\perp(A_2)$. Hence, if u_i is in a non-variable term s_k in A_1 then v_i is in a non-variable term t_k in A_2 and therefore if s_k is a non-variable term then t_k is a corresponding non-variable term. Then according to Lemma 8, there exists a substitution θ such that $p(s_1, .., s_n)\theta = p(t_1, .., t_n)$ and therefore $A_1 \succeq A_2$. $\square$

Theorem 13 *The mapping function $\pi_\perp : \mathcal{A}_\perp \to \Pi_\perp$ as defined in Definition 67 is an order-isomorphism.*

Proof. First we show that the mapping function $\pi_\perp$ is onto. Let π be a partition in $\Pi_\perp$. We show that there is an atom A in $\mathcal{A}_\perp$ such that $\pi_\perp(A) = \pi$. Let $\mathcal{A}_\perp$ and $a(\overrightarrow{\perp})$ be as defined in Definition 62 and $A = p(t_1, t_2, \ldots, t_n)$ be an atom in $\mathcal{A}_\perp$ such that for each block B in π and for each $\{i, j\} \subseteq B$, variables v_i and v_j in some non-variable term(s) in A are the same, where i and j correspond to the ith and jth variable positions in $a(\overrightarrow{\perp})$, and other terms in A are distinct variables. Then according to Definition 67 we have $\pi_\perp(A) = \pi$ and therefore $\pi_\perp$ is onto. Moreover, according to Theorem 12 and Definition 26, the mapping function $\pi_\perp$ is order-embedding. Then according to Definition 26, $\pi_\perp$ is an order-isomorphism. $\square$

The proposition below follows directly from Theorem 13 and Proposition 7.

Proposition 15 *The mapping function $\pi_\perp : \mathcal{A}_\perp \to \Pi_\perp$ as defined in Definition 67 is a lattice isomorphism and lattices $\langle \Pi_\perp, \leq \rangle$ and $\langle \mathcal{A}_\perp, \succeq \rangle$ are two isomorphic lattices.*

The proposition below follows directly from Proposition 15 and Proposition 11.

Proposition 16 *Lattices $\langle \Pi_\perp, \leq \rangle$ and $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ are two isomorphic lattices.*

4.4 Related work and discussion

The lattice structure of atomic formulas has been studied by Reynolds [Rey69]. He showed that the set of all equivalence classes of atoms (under alphabetic variation), augmented by adding a 'universal formula' and a 'null formula', form a lattice. He also showed that every atom has a finite decending (or ascending) chain, i.e. a finite set of downward (or upward) covers. He also described an efficient algorithm for computing the least general generalisation (*lgg*) of two atoms. Plotkin [Plo69] described a similar *lgg* algorithm for atoms. He also extended the investigation to clauses ordered by θ-subsumption and studied the lattice properties of clauses. However, Plotkin has shown that, unlike for atoms, infinite decending chains exist for clauses [Plo71]. In this chapter we have shown that the lattice of bounded subsumption shares several properties with the lattice of atoms. This includes the existence of a finite set of covers. We eventually show that the lattice of bounded subsumption can be mapped to an atomic lattice and therefore the properties of the atomic lattice, e.g. those described by Reynolds [Rey69], are applicable here. Note that unlike the lattice considered by Reynolds, the lattice of bounded subsumption (and its equivalent atomic lattice) do not require a special formula (i.e. null formula) to represent the least element of the lattice and the bottom clause (or its atomic representation) is the least element of the lattice.

A subsumption relation for ordered clauses (i.e. ordered subsumption) is studied in [KOHH06]. Some of differences between ordered subsumption and the subsumption orders considered in this thesis (i.e. sequential subsumption and subsumption relative to a bottom clause) were discussed in Section 3.5. It is shown [KOHH06] that (i) the least generalisation of two ordered clauses does not exist under ordered subsumption (and therefore ordered subsumption does not form a lattice) and (ii) the ordered subsumption testing is NP-complete. Another related subsumption order is s-subsumption which was defined for simple sequences in SeqLog [LD04, Lee06]. It is shown [Lee06] that time complexity of s-subsumption is polynomial but the general SeqLog subsumption among complex sequences is NP-complete. It is also shown that lgg does not exist for any pair of simple sequences or complex sequences and subsumption for simple or complex sequences do not form lattices.

The sequential subsumption on ordered clauses defined in this thesis (Section 3.3) is similar to s-subsumption on simple sequences and by similar proofs, the results for s-subsumption (shown in [Lee06]) can also be shown for sequential subsumption, i.e. time complexity of sequential subsumption is polynomial and lgg does not exist for sequential subsumption. However, we have shown that each pair of ordered clauses have a most general specialisation and a least general generalisation under subsumption relative to a bottom clause (i.e. $lgg_\perp$ and $mgs_\perp$) and that the bounded subsumption forms a lattice. We have also shown that subsumption testing relative to a bottom clause can be mapped to atomic subsumption testing. An atomic subsumption testing can be reduced to a unification problem which can be decided in linear time [GL85].

4.5 Summary

In this chapter we have defined the most general specialisation and the least general generalisation for the subsumption order relative to the bottom clause (i.e. $mgs_\perp$ and $lgg_\perp$) and we have shown that the bounded subsumption forms a lattice. We have also defined downward covers for the bounded subsumption and have shown that, unlike for θ-subsumption, a finite set of downward covers exists. In this chapter we have also defined a mapping between the lattice of bounded subsumption and an atomic lattice and we have shown that these two lattices are isomorphic. We also showed that the atomic lattice is isomorphic to a lattice of partitions. Hence, the lattice of bounded subsumption can be encoded as a lattice of partitions and therefore clause refinement can be mapped to partition refinement. This encoding will be discussed further in the next chapter.

Chapter 5

Encoding and refinement operators for bounded subsumption

In this chapter we study encoding and refinement operators for the subsumption order relative to $\bot$. We show that, unlike for the general subsumption order, ideal refinement operators exist for the lattice of bounded subsumption, i.e. $\langle \overrightarrow{\mathcal{L}}_{\bot}, \succeq_{\bot} \rangle$. In this chapter we also study an encoding of the elements of this lattice which reflects the structure of the lattice and can be exploited by the refinement operators and algorithms. This encoding and a refinement operator ρ_1 for bounded subsumption are introduced in Section 5.1. In Section 5.2, we show that ρ_1 is ideal for the lattice $\langle \overrightarrow{\mathcal{L}}_{\bot}, \succeq_{\bot} \rangle$. Each clause $\overrightarrow{C}$ in this lattice is encoded by a tuple $\langle K, \theta \rangle$, where K is a set of indexes from the bottom clause and θ is a variable subsumption which maps variables between $\overrightarrow{C}$ and $\overrightarrow{\bot}_v$. In Section 5.3 we study the mapping and the morphism between $\langle \overrightarrow{\mathcal{L}}_{\bot}, \succeq_{\bot} \rangle$ and the lattice of the encoding tuples $\langle K, \theta \rangle$. In Section 5.4, we study alternative encoding and refinement operators for bounded subsumption. Related work is discussed in Section 5.5. Section 5.6 summarises the chapter.

5.1 Mapping function c and refinement operator ρ_1

It is known that when a full Horn clause language and the general subsumption order are considered, there exist no ideal refinement operators [vdLNC94]. However, if $\langle \mathcal{L}, \geq \rangle$ is a quasi-order, $\mathcal{L}$ is finite and $\geq$ is decidable, then there exists an ideal refinement operator for $\langle \mathcal{L}, \geq \rangle$ [NCdW97]. Given the finiteness of $\overrightarrow{\mathcal{L}}_\perp$, one could expect the existence of ideal refinement operators for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. As shown in Chapter 3, Progol's refinement ρ is weakly complete but not complete and therefore ρ cannot be an ideal refinement operator. In this section we define a refinement operator ρ_1 and show that ρ_1 is ideal for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. The main difference between ρ and ρ_1 is that, unlike ρ which can only add a new literal at the end of the clause, ρ_1 can add a new literal anywhere in the clause (e.g. in the middle). In both cases the new literal is a generalisation of a literal from the bottom clause. In ρ an incremental index k is maintained and for each value of k it is decided whether to include a generalisation of the k-th literal of the bottom clause. In the case of ρ_1, instead of the index k, a set K is maintained which contains the indexes of the literals from the bottom clause. In ρ_1, at each step of the refinement a new index can be added to K representing a literal from any position in the bottom clause. However, the order of literals of a clause in $\overrightarrow{\mathcal{L}}_\perp$ should follow the same order as literals in the bottom clause. In Progol's refinement operator ρ, in addition to the indexes k a substitution θ is maintained for each clause in order to decode the clause from $\overrightarrow{\perp}$. In this setting, substitution θ maps variables from $\overrightarrow{C}$ to the variables of $\overrightarrow{\perp}$. The decoding, therefore, involves inverse substitution θ^{-1}. This can be achieved by maintaining the position of variables when the substitution θ is constructed [NCdW97]. However, in the encoding used for the refinement operator ρ_1, a substitution θ maps variables from $\overrightarrow{\perp_v}$ (see Definition 53) to the variables of $\overrightarrow{C}$. Distinct variables in $\overrightarrow{\perp_v}$ also represent the variable positions and therefore no extra information about variable positions is needed. Then, variable binding is done by partitions over distinct variables. In the encoding described for the refinement operator ρ_1 in this section, variable partitions are represented by variable substitution θ and the presence of literals is explicitly represented by a set of corresponding indexes K. This representation can also be explained by the morphism between the lattice of the subsumption order relative to a bottom clause and the partition lattice. This is discussed further in Section 5.3. The following mapping function c maps a tuple $\langle K, \theta \rangle$ into an ordered clause in $\overrightarrow{\mathcal{L}}_\perp$.

Definition 68 (Mapping function c**)** *Let $\overrightarrow{\mathcal{L}}_\perp$ and $\overrightarrow{\perp_v}$ be as defined in Definition 53, $\theta_\perp$ be as defined in Definition 54, n be the number of literals in $\overrightarrow{\perp_v}$, $\mathcal{K}$ be the power set of $\{1, \ldots, n\}$ and $K \in \mathcal{K}$. Let θ be a variable substitution in Θ, where $\Theta = \{\theta | \theta \subseteq \theta_\perp$*

and if $\{v_j/u, u/v_i\} \subseteq \theta$ *then* $v_j/v_i \in \theta\}$*. The mapping function* $c : \mathcal{K} \times \Theta \to \overrightarrow{\mathcal{L}}_\perp$ *is defined as follows:*

$$c(\langle K, \theta\rangle) = (\bigvee_{i=1}^{n} l_i \text{ where } i \in K \text{ and } l_i \text{ is the ith literal of } \overrightarrow{\perp_v})\theta.$$

In this definition $\theta_\perp$ is a substitution relative to $\perp$ and Θ is the set of all subsets of $\theta_\perp$ containing transitive bindings.

Note that (as in Theorem 4) the above disjunction notion with indexes from $i = 1$ to n means that the literals l_i of $c(\langle K, \theta\rangle)$ follow the same order as literals in $\overrightarrow{\perp_v}$.

Example 30 *Let* $\overrightarrow{\perp}$ *be the bottom clause in Example 7.* $\overrightarrow{\perp_v}$ *is obtained from* $\overrightarrow{\perp}$ *by populating all variable positions with new and distinct variables:*

$$\begin{aligned}\overrightarrow{\perp_v} = mult(V_1, V_2, V_3) \quad \leftarrow \quad & dec(V_4, V_5), plus(V_6, V_7, V_8), plus(V_9, V_{10}, V_{11}), \\ & mult(V_{12}, V_{13}, V_{14}), mult(V_{15}, V_{16}, V_{17}).\end{aligned}$$

Let $K = \{1, 2, 5\}$*,* $\theta = \{V_4/V_1, V_{12}/V_5\}$ *then* $c(\langle K, \theta\rangle)$ *can be defined as follows:*

$$\overrightarrow{C} = c(\langle K, \theta\rangle) = mult(V_1, V_2, V_3) \quad \leftarrow \quad dec(V_1, V_5), mult(V_5, V_{13}, V_{14}).$$

$\diamond$

The mapping function c maps a tuple $\langle K, \theta\rangle$ into an ordered clause $\overrightarrow{C}$ in $\overrightarrow{\mathcal{L}}_\perp$. This mapping function is also defined such that the literals in $\overrightarrow{C}$ follow the same order as literals in $\overrightarrow{\perp}$. This condition is important for the refinement operator ρ_1 which is intended to be complete for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp\rangle$.

The refinement operator ρ_1 is similar to Progol's ρ in the sense that it works by adding a new literal which is a generalisation of a literal from the bottom clause. However, ρ can only add a new literal at the end of the clause, while ρ_1 can add a new literal anywhere in the clause. Moreover, ρ generalises a literal from $\perp$ by variable splitting (see Section 3.2), while ρ_1 specialises a literal from $\overrightarrow{\perp_v}$ by a valid variable binding with respect to $\perp$ (defined by substitutions relative to $\perp$, i.e. $\theta_\perp$).

Definition 69 (ρ_1) *Let $\overrightarrow{\bot}$ and $\overrightarrow{\mathcal{L}}_\bot$ be as defined in Definition 53, $\overrightarrow{C}$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\bot$, n be the number of literals in $\overrightarrow{\bot}$, k be a natural number, $1 \leq k \leq n$, $\overrightarrow{\bot_v}$, Θ, $\mathcal{K}$ and the mapping function c be defined as in Definition 68. Let $K \in \mathcal{K}$, $\theta \in \Theta$, $\overrightarrow{C} = c(\langle K, \theta\rangle)$ then $\langle \overrightarrow{C'}, K', \theta'\rangle$ is in $\rho_1(\langle \overrightarrow{C}, K, \theta\rangle)$ if and only if $\overrightarrow{C'} = c(\langle K', \theta'\rangle)$ and either*

1. *$K' = K \cup \{k\}$, $k \notin K$ and $\theta' = \theta$ or*
2. *$K' = K$, $\theta' = \theta\{y'/x'\}$ and $\{y'/x'\} \in \Theta$ where x' and y' are distinct variables in the k_1th and k_2th literals of $\overrightarrow{\bot_v}$ respectively and k_1th and k_2th are in K'.*

In Definition 69, ρ_1 adds a most general literal from $\overrightarrow{\bot_v}$ which has not been added before (choice 1) or it applies an elementary variable substitution such that the clause subsumes $\overrightarrow{\bot}$ (choice 2). The variable substitution θ in an encoding tuple $\langle K, \theta\rangle$ defines a set of equivalences classes over the distinct variables in $\overrightarrow{\bot_v}$. The second choice in the definition of ρ_1 merges two equivalence classes by applying the variable binding $\{y'/x'\}$ on θ.

Example 31 *Let $\overrightarrow{\bot_v}$, K, θ and $\overrightarrow{C}$ be as defined in Example 30. Then $\langle \overrightarrow{C'}, K', \theta'\rangle$ is in $\rho_1(\langle \overrightarrow{C}, K, \theta\rangle)$ and (a) $K' = \{1, 2, 5, 6\}$, $\theta' = \{V_4/V_1, V_{12}/V_5\}$ and*

$$\begin{aligned}\overrightarrow{C'} = c(\langle K', \theta'\rangle) = \mathit{mult}(V_1, V_2, V_3) \quad \leftarrow \quad & \mathit{dec}(V_1, V_5), \mathit{mult}(V_5, V_{13}, V_{14}), \\ & \mathit{mult}(V_{15}, V_{16}, V_{17})\end{aligned}$$

is a possible clause if choice 1 in ρ_1 is selected and (b) $K' = \{1, 2, 5\}$, $\theta' = \{V_4/V_1, V_{12}/V_5, V_{13}/V_2\}$ and

$$\overrightarrow{C'} = c(\langle K', \theta'\rangle) = \mathit{mult}(V_1, V_2, V_3) \quad \leftarrow \quad \mathit{dec}(V_1, V_5), \mathit{mult}(V_5, V_2, V_{14})$$

is another possible clause if choice 2 in ρ_1 is selected. These two example applications of refinement operator ρ_1 are shown in Figures 5.1a and 5.1b. $\diamond$

C'	θ'	K'
$\square$	$\emptyset$	$\emptyset$
$mult(V_1, V_2, V_3) \leftarrow$	$\emptyset$	$\{1\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_4, V_5)$	$\emptyset$	$\{1, 2\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5)$	$\{V_4/V_1\}$	$\{1, 2\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_{12}, V_{13}, V_{14})$	$\{V_4/V_1\}$	$\{1, 2, 5\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_5, V_{13}, V_{14})$	$\{V_4/V_1, V_{12}/V_5\}$	$\{1, 2, 5\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_5, V_2, V_{14})$ $mult(V_{15}, V_{16}, V_{17})$	$\{V_4/V_1, V_{12}/V_5\}$	$\{1, 2, 5, 6\}$

(a)

C'	θ'	K'
$\square$	$\emptyset$	$\emptyset$
$mult(V_1, V_2, V_3) \leftarrow$	$\emptyset$	$\{1\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_4, V_5)$	$\emptyset$	$\{1, 2\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5)$	$\{V_4/V_1\}$	$\{1, 2\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_{12}, V_{13}, V_{14})$	$\{V_4/V_1\}$	$\{1, 2, 5\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_5, V_{13}, V_{14})$	$\{V_4/V_1, V_{12}/V_5\}$	$\{1, 2, 5\}$
$mult(V_1, V_2, V_3) \leftarrow dec(V_1, V_5), mult(V_5, V_{13}, V_{14})$	$\{V_4/V_1, V_{12}/V_5, V_{13}/V_2\}$	$\{1, 2, 5\}$

(b)

Figure 5.1: Two example applications of refinement operator ρ_1.

5.2 Idealness of refinement operator ρ_1

In the following we show that ρ_1 is ideal for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. The completeness proof below is similar to the completeness proof for Laird's refinement operator [vdL95] adapted for subsumption order relative to $\perp$.

Lemma 9 *Let $\overrightarrow{C}$, $\overrightarrow{D}$ be two ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C}\theta = \overrightarrow{D}$ for some substitution $\theta \subseteq \theta_\perp$. Then, there exists a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$.*

Proof. Suppose $\overrightarrow{C}$, $\overrightarrow{D}$ are ordered clauses and $\overrightarrow{C}\theta = \overrightarrow{D}$. This lemma is then a special case of Theorem 7 and there is a finite chain of downward covers of Type 1 (Lemma 5) from $\overrightarrow{C}$ to $\overrightarrow{D}$ involving substitution θ. Thus, there exists a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$ by repeatedly selecting choice 2 in Definition 69. $\square$

Lemma 10 *Let $\overrightarrow{C}$, $\overrightarrow{D}$ be two ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that $\overrightarrow{C}$ is a subsequence of $\overrightarrow{D}$. Then, there exists a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$.*

Proof. The proof is by induction on i the number of literals in $\overrightarrow{D}$ but not in $\overrightarrow{C}$. If $i = 0$ then $\overrightarrow{C} = \overrightarrow{D}$, and the empty chain satisfies the lemma. Assume for some j, $0 \leq j < i$, the lemma is true. This implies that there is a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{C}_j$ such that $\overrightarrow{C}_j$ is $\overrightarrow{C}$ with j literals inserted such that $\overrightarrow{C}_j$ is a subsequence of $\overrightarrow{D}$. We show that there is a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{C}_{j+1}$. Let l be the leftmost literal in $\overrightarrow{D}$ which is not in $\overrightarrow{C}_j$. Given that $\overrightarrow{D} \in \overrightarrow{\mathcal{L}}_\perp$ we can assume that l is mapped to the k-th literal of $\overrightarrow{\perp}$. We consider the following two cases: (a) if l is a most general literal with respect to $\overrightarrow{C}_j$, then l is the k-th literal of $\overrightarrow{\perp_v}$ and using choice 1 in the definition of ρ_1, $\langle \overrightarrow{C}_{j+1}, K', \theta \rangle \in \rho_1(\langle \overrightarrow{C}_j, K, \theta \rangle)$, where $K' = K \cup \{k\}$, (b) otherwise there is a most general literal l' such that $l'\theta' = l$. In this case, first using choice 1 in the definition of ρ_1, $\langle \overrightarrow{C'}_{j+1}, K', \theta \rangle \in \rho_1(\langle \overrightarrow{C}_j, K, \theta \rangle)$ and then according to Lemma 9 (and using choice 2 in the definition of ρ_1), $\langle \overrightarrow{C}_{j+1}, K', \theta'' \rangle \in \rho_1^*(\langle \overrightarrow{C'}_{j+1}, K', \theta \rangle)$, where $K' = K \cup \{k\}$ and $\theta'' = \theta\theta'$. Thus, in both cases (a) and (b), there exists a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{C}_{j+1}$ and this completes the proof. □

Theorem 14 *ρ_1 is complete for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$.*

Proof. Let $\overrightarrow{C}$, $\overrightarrow{D}$ be two ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$ such that, for some $\theta \subseteq \theta_\perp$, $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. We need to show that there is ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$. If we define $\overrightarrow{E} = \overrightarrow{C}\theta$ then $\overrightarrow{E}$ and $\overrightarrow{C}$ satisfy Lemma 9, hence there is a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{E}$. $\overrightarrow{E}$ is a subsequence of $\overrightarrow{D}$ relative to $\perp$ and according to Lemma 10, there is a ρ_1-chain from $\overrightarrow{E}$ to $\overrightarrow{D}$. Thus, there is a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$ via $\overrightarrow{E}$. □

In the following we show the properness and the idealness of ρ_1 for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$.

Lemma 11 *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}_\perp$. If $\overrightarrow{C} \sim_\perp \overrightarrow{D}$, then $\overrightarrow{C}$ and $\overrightarrow{D}$ are alphabetical variants.*

Proof. Suppose $\overrightarrow{C} \sim_\perp \overrightarrow{D}$, then we have $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ and $\overrightarrow{D} \succeq_\perp \overrightarrow{C}$. Thus, there are substitutions θ_1 and θ_2 in $\theta_\perp$ such that $\overrightarrow{C}\theta_1$ is a subsequence of $\overrightarrow{D}$ and $\overrightarrow{D}\theta_2$ is a subsequence of $\overrightarrow{C}$. Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_l$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$. Hence, there are strictly increasing selection functions s_1 and s_2 such that for each $(i, j) \in s_1$, $L_i\theta_1 = M_j$ and for each $(i, j) \in s_2$, $M_i\theta_2 = L_j$. Given that s_1 and s_2 are strictly increasing functions, there is a one-to-one mapping between literals of $\overrightarrow{C}$ and $\overrightarrow{D}$ such that $m = n$, $L_i\theta_1 = M_i$ and $M_i\theta_2 = L_i$. Therefore it holds that $\overrightarrow{C}\theta_1 = \overrightarrow{D}$ and $\overrightarrow{D}\theta_2 = \overrightarrow{C}$. Hence, $\overrightarrow{C}$ and $\overrightarrow{D}$ are alphabetical variants. □

Lemma 12 *Let $\mathcal{K}$ and Θ and mapping function c be defined as in Definition 68 and $K, \{k\} \in \mathcal{K}$ such that $k \notin K$ and $\theta \in \Theta$. Then, $c(\langle K \cup \{k\}, \theta \rangle) \succ_\perp c(K, \theta)$.*

Proof. Suppose $c(\langle K \cup \{k\}, \theta \rangle) \not\succ_\perp c(K, \theta)$. We know from Theorem 17 that $c(\langle K \cup \{k\}, \theta \rangle) \succeq_\perp c(K, \theta)$, and therefore $c(\langle K \cup \{k\}, \theta \rangle) \sim_\perp c(K, \theta)$. According to Lemma 11, $c(\langle K \cup \{k\}, \theta \rangle)$ and $c(K, \theta)$ must be alphabetical variants, contradicting $k \notin K$. Thus, $c(\langle K \cup \{k\}, \theta \rangle) \succ_\perp c(K, \theta)$. □

Lemma 13 *Let $\mathcal{K}$ and Θ, $\perp_v$ and mapping function c be defined as in Definition 68 and $K \in \mathcal{K}$, $\{y/x\}, \theta \in \Theta$ where x and y are distinct variables in the k_1th and k_2th literals of $\overrightarrow{\perp_v}$ respectively and k_1th and k_2th are in K. Then, $c(\langle K, \theta\{y/x\} \rangle) \succ_\perp c(K, \theta)$.*

Proof. Suppose $c(\langle K, \theta\{y/x\} \rangle) \not\succ_\perp c(K, \theta)$. We know from Theorem 17 that $c(\langle K, \theta\{y/x\} \rangle) \succeq_\perp c(K, \theta)$, and therefore $c(\langle K, \theta\{y/x\} \rangle) \sim_\perp c(K, \theta)$. According to Lemma 11, $c(\langle K, \theta\{y/x\} \rangle)$ and $c(K, \theta)$ must be alphabetical variants. Thus, $\{y/x\}$ must be a renaming subsumption, i.e. x is either equal to y or it does not occur in $c(K, \theta)$, contradicting the assumption. Thus, $c(\langle K, \theta\{y/x\} \rangle) \succ_\perp c(K, \theta)$. □

Theorem 15 ρ_1 *is proper for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$.*

Proof. If $\langle C', \theta', K' \rangle \in \rho_1(\langle C, \theta, K \rangle)$ is generated by choice 1 in the definition of ρ_1, then $\overrightarrow{C} \succ_\perp \overrightarrow{D}$ follows from Lemma 12. If it is generated by choice 2 in the definition of ρ_1, then $\overrightarrow{C} \succ_\perp \overrightarrow{D}$ follows from Lemma 13. □

Theorem 16 ρ_1 *is ideal for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$.*

Proof. Locally finiteness follows from the definition of ρ_1 and the fact that there are finite number of literals and variables in $\perp$. Completeness and properness were proved in Theorem 14 and Theorem 15 respectively. □

5.3 Lattice isomorphism of mapping function c

In this section we study the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and a lattice of the encoding tuples $\langle K, \theta \rangle$, used in the mapping function c (i.e. $\langle \mathcal{K} \times \Theta, \subseteq \rangle$). As mentioned in Section 5.1, the variable substitution θ in an encoding tuple $\langle K, \theta \rangle$ defines a set of equivalence classes over the distinct variables in $\overrightarrow{\perp_v}$. The presence of literals from $\overrightarrow{\perp_v}$ is explicitly represented by a set of corresponding indexes K. However, as shown in Section 4.3, the presence of literals can also be implicitly represented by the presence of variables in θ. The morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and the partitions lattice was shown in Chapter 4. It is known (e.g. see Theorem 4.11 in [SB81]) that any partition on a set induces an equivalence relation on it and conversely, any equivalence relation induces a partition.

Proposition 17 ([SB81]) *Let $\langle \Pi_n, \leq \rangle$ be as defined in Definition 64 and E_n be the set of all equivalence relations on $\{1, 2, \ldots, n\}$. The mapping function $\varepsilon : \Pi_n \rightarrow E_n$ defined as $\varepsilon(\pi) = \{\langle i, j \rangle | \{i, j\} \subseteq B$ for some B in $\pi\}$ is a lattice isomorphism and lattices $\langle \Pi_n, \leq \rangle$ and $\langle E_n, \subseteq \rangle$ are two isomorphic lattices.*

The morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \subseteq \rangle$ can be explained by Proposition 17 and the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and the partitions lattice (see Proposition 16). However, in this chapter we give a direct proof for the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \subseteq \rangle$.

In the following we first define the order relation for the encoding tuples $\langle K, \theta \rangle$, used in the mapping function c. Then, we show that the mapping function c is order-embedding.

Definition 70 *Let Θ be defined as in Definition 68 and $\theta_1, \theta_2 \in \Theta$. $\theta_1 \leq \theta_2$ if there exists a substitution $\theta \in \Theta$ such that $\theta_1\theta = \theta_2$.*

Definition 71 *Let $\mathcal{K}$ and Θ be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. $\langle K_1, \theta_1 \rangle \subseteq \langle K_2, \theta_2 \rangle$ if and only if $K_1 \subseteq K_2$ and $\theta_1 \leq \theta_2$. $\langle K_1, \theta_1 \rangle \sim \langle K_2, \theta_2 \rangle$ if and only if $\langle K_1, \theta_1 \rangle \subseteq \langle K_2, \theta_2 \rangle$ and $\langle K_2, \theta_2 \rangle \subseteq \langle K_1, \theta_1 \rangle$.*

Theorem 17 *Let $\mathcal{K}$ and Θ and function c be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. $c(\langle K_1, \theta_1 \rangle) \succeq_\perp c(\langle K_2, \theta_2 \rangle)$ if and only if $\langle K_1, \theta_1 \rangle \subseteq \langle K_2, \theta_2 \rangle$.*

Proof. $\Rightarrow$: Let $\overrightarrow{C}$, $\overrightarrow{D}$ be ordered clauses such that $\overrightarrow{C} = c(\langle K_1, \theta_1\rangle)$ and $\overrightarrow{D} = c(\langle K_2, \theta_2\rangle)$. Assume $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$, then according to Theorem 14 there is a ρ_1-chain from $\overrightarrow{C}$ to $\overrightarrow{D}$. Let this chain be $\overrightarrow{C} = \overrightarrow{C'}_0 \succeq_\perp \overrightarrow{C'}_1 \succeq_\perp \cdots \succeq_\perp \overrightarrow{C'}_m = \overrightarrow{D}$ where $\langle \overrightarrow{C'}_{i+1}, K'_{i+1}, \theta'_{i+1}\rangle \in \rho_1(\langle \overrightarrow{C'}_i, K'_i, \theta'_i\rangle)$, $0 \leq i < m$. According to the definition of ρ_1, in each refinement step either 1) $K'_i \subseteq K'_{i+1}$ and $\theta'_{i+1} = \theta'_i$ or 2) $K'_{i+1} = K'_i$ and $\theta'_i \leq \theta'_{i+1}$. Then it is always the case that $K'_i \subseteq K'_{i+1}$ and $\theta'_i \leq \theta'_{i+1}$, where $K'_0 = K_1, K'_m = K_2, \theta'_0 = \theta_1, \theta'_m = \theta_2$. Thus, $K_1 \subseteq K_2$ and $\theta_1 \leq \theta_2$ and therefore $\langle K_1, \theta_1\rangle \subseteq \langle K_2, \theta_2\rangle$.

$\Leftarrow$: Let $\overrightarrow{C} = c(K_1, \theta_1) = (\bigvee l_i | i \in K_1)\theta_1$ and $\overrightarrow{D} = c(K_2, \theta_2) = (\bigvee l_j | j \in K_2)\theta_2$ such that $K_1 \subseteq K_2$ and $\theta_1 \leq \theta_2$. According to Definition 71, $\theta_1\theta = \theta_2$ for some substitution $\theta \in \Theta$. Then, given $K_1 \subseteq K_2$, for every literal $l_i\theta_1\theta$ from $\overrightarrow{C}\theta$, we have a literal $l_j\theta_2 = l_i\theta_1\theta$ from $\overrightarrow{D}$ and these literals are both mapped to the i-th literal from $\overrightarrow{\perp_v}$ (Definition 68). Thus, $\overrightarrow{C}\theta$ is a subsequence of $\overrightarrow{D}$. Moreover, $\theta \in \Theta$ and therefore $\theta \subseteq \theta_\perp$. Hence, $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$. □

According to Theorem 17, the mapping function c is an order-embedding. The following theorem shows that c is also an order-isomorphism.

Theorem 18 *The mapping function $c : \mathcal{K} \times \Theta \rightarrow \overrightarrow{\mathcal{L}}_\perp$ as defined in Definition 68 is an order-isomorphism.*

Proof. First we show that the mapping function c is onto. Let $\overrightarrow{C}$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\perp$, then there exist substitution θ and selection function s which maps literals of $\overrightarrow{C}\theta$ to equivalent literals from $\overrightarrow{\perp}$. From Definition 68 we have $\overrightarrow{\perp_v}\theta_v = \overrightarrow{\perp}$ and therefore $\overrightarrow{C}\theta \sqsubseteq \overrightarrow{\perp_v}\theta_v$ and this implies $\overrightarrow{C} \sqsubseteq \overrightarrow{\perp_v}\theta_v\theta^{-1}$. Thus, $\overrightarrow{C}$ can be defined as $\overrightarrow{C} = c(K, \theta') = (\bigvee l_i | i \in K)\theta'$, where $\theta' = \theta_v\theta^{-1}$ and K is the range of the selection function s. Hence, the mapping function c is onto. Moreover, according to Theorem 17, the mapping function c is an order-embedding. Then according to Definition 26, c is an order-isomorphism. □

The proposition below follows directly from Theorem 18.

Proposition 18 *Let $\mathcal{K}$ and Θ and mapping function c be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. $c(K, \theta) \sim_\perp c(K', \theta')$ if and only if $\langle K, \theta \rangle \sim \langle K', \theta' \rangle$.*

According to Theorem 6 $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ is a lattice. The proposition below follows directly from Theorem 18 and Remark 7.

Proposition 19 *The mapping function $c : \mathcal{K} \times \Theta \to \overrightarrow{\mathcal{L}}_\perp$ as defined in Definition 68 is a lattice isomorphism and lattices $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \subseteq \rangle$ are two isomorphic lattices.*

The proposition below follows from c being a lattice isomorphism.

Proposition 20 *Let $\mathcal{K}$ and Θ and mapping function c be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. Mapping c is join-preserving and meet-preserving, that is:*

1. $lgg_\perp(c(\langle K_1, \theta_1 \rangle, c(\langle K_2, \theta_2 \rangle) = c(\langle K_1, \theta_1 \rangle \cap \langle K_2, \theta_2 \rangle)$
2. $mgs_\perp(c(\langle K_1, \theta_1 \rangle, c(\langle K_2, \theta_2 \rangle) = c(\langle K_1, \theta_1 \rangle \cup \langle K_2, \theta_2 \rangle)$

According to Proposition 20, the least general generalisation ($lgg_\perp$) and the most general specialisation ($mgs_\perp$) for $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ can be defined based on the join and the meet operations for $\langle \mathcal{K} \times \Theta, \subseteq \rangle$. The morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, lgg_\perp, mgs_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \cap, \cup \rangle$ is important from a practical point of view. The construction of the least general generalisation (lgg) of clauses in the general subsumption order is inefficient as the cardinality of the lgg of two clauses can grow very rapidly (see Section 4.2). On the other hand, efficient operators can be implemented for least generalisation and greatest specialisation in the subsumption order relative to a bottom clause. For example, with Plotkin's Relative Least General Generalisation (RLGG), clause length grows exponentially in the number of examples [Plo71]. Hence, an ILP system like Golem [MF90] which uses RLGG is constrained to ij-determinacy to guarantee polynomial-time construction. However, the determinacy restrictions make an ILP system inapplicable in many key application areas, including the learning of chemical properties from atom and bond descriptions. On the other hand, a variant of Plotkin's Relative RLGG which does not need the determinacy restrictions can be designed based on subsumption with respect to a bottom clause. This idea is the basis of Asymmetric Relative Minimal Generalisations (or ARMGs) relative to a bottom clause where the clause length is bounded by the length of the initial bottom clause. ARMGs, therefore do not need the determinacy restrictions used in Golem. ARMGs have been implemented in an ILP system called ProGolem [MSTN10a] which combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. ARMGs are discussed further in Chapter 7.

5.4 Alternative forms of bounded subsumption, encoding and refinement operators

The purpose of Chapters 3 and 4 was to characterise Progol's refinement and the subsumption lattice which is searched by a Progol-like ILP system, i.e. the bounded subsumption lattice. We defined (sequential) subsumption order relative to $\perp$ and studied the properties of this special case of subsumption. In this section we show how other forms of subsumption relative to $\perp$ can be defined by using different conditions on the selection functions which define subsequences. For example we show how the first type of incompleteness in Progol's refinement operator (see Chapter 3) can be addressed by relaxing conditions of subsumption order relative to a bottom clause. As demonstrated in section 3.2, the first type of Progol's refinement incompleteness is due to the choice of ordering of literals in $\perp$ and the fact that clauses are considered as subsequences of $\perp$. This condition was embedded in the definitions of the subsumption order relative to a bottom clause in Chapter 3 and also the refinement operator ρ_1 in this chapter. However, more relaxed conditions can be defined for subsumption and refinement operators relative to $\perp$. Note that in the definitions and theorems for bounded subsumption, in particular the encoding described in this chapter, we only needed to assume that the selection functions are injective so that we can encode every literal of a clause by a k index from $\perp$. A less restricted ordering can therefore be defined by using a selection function which is injective rather than strictly increasing. Hence, instead of using subsequence we use injective subset which is defined as follows.

Definition 72 (Injective subset) *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \cdots \vee L_l$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \cdots \vee M_m$ be ordered clauses. $\overrightarrow{C}$ is an injective subset of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \sqsubseteq^i \overrightarrow{D}$, if*

$$\begin{tikzcd}[column sep=small]
\overrightarrow{C} = & p(x,y) \arrow[d] & \vee & r(x,y) \arrow[d, rr] & \vee & r(y,x) \arrow[d, rr] & & \\
\overrightarrow{B} = & p(x,y) & \vee & q(x,y) & \vee & r(x,y) & \vee & r(y,x)
\end{tikzcd}$$

(a)

$$\begin{tikzcd}[column sep=small]
\overrightarrow{D} = & p(x,y) \arrow[d] & \vee & r(y,x) \arrow[d, rrrr] & \vee & r(x,y) \arrow[d] & & \\
\overrightarrow{B} = & p(x,y) & \vee & q(x,y) & \vee & r(x,y) & \vee & r(y,x)
\end{tikzcd}$$

(b)

Figure 5.2: Comparison between subsequence and injective subset **(a)** $\vec{C}$ is a subsequence and also an injective subset of $\vec{B}$ **(b)** $\vec{D}$ is not a subsequence of $\vec{B}$, however, it is an injective subset of $\vec{B}$.

there exists an injective selection function $s \subseteq \{1, \ldots, l\} \times \{1, \ldots, m\}$ such that for each $(i, j) \in s$, $L_i = M_j$.

Example 32 *In Figure 5.2, $\overrightarrow{C}$ is a subsequence of $\overrightarrow{B}$ because there exists increasing selection function $s_1 = \{(1,1),(2,3),(3,4)\}$ which maps literals from $\overrightarrow{C}$ to equivalent literals from $\overrightarrow{B}$. s_1 is also an injective selection function and $\overrightarrow{C}$ is therefore an injective subset of $\overrightarrow{B}$. $\overrightarrow{D}$ is not a subsequence of $\overrightarrow{B}$ because an increasing selection function does not exist for $\overrightarrow{D}$ and $\overrightarrow{B}$. However, $\overrightarrow{D}$ is an injective subset of $\overrightarrow{B}$ because there exists an injective selection function $s_2 = \{(1,1),(2,4),(3,3)\}$ which maps literals from $\overrightarrow{D}$ to equivalent literals from $\overrightarrow{B}$.* $\diamond$

Definition 73 (Injective subsumption) *Let $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses. We say $\overrightarrow{C}$ is an injective generalisation of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \succeq^i \overrightarrow{D}$, if there exists a substitution θ such that $\overrightarrow{C}\theta$ is an injective subset of $\overrightarrow{D}$. $\overrightarrow{C}$ is a proper injective generalisation of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \succ^i \overrightarrow{D}$, if $\overrightarrow{C} \succeq^i \overrightarrow{D}$ and $\overrightarrow{D} \not\succeq^i \overrightarrow{C}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ are equivalent with respect to injective subsumption, denoted by $\overrightarrow{C} \sim^i \overrightarrow{D}$, if $\overrightarrow{C} \succeq^i \overrightarrow{D}$ and $\overrightarrow{D} \succeq^i \overrightarrow{C}$.*

Example 33 *Let $\overrightarrow{B} = p(X_1, X_1) \vee q(X_1, Y_1) \vee r(X_1, Y_1) \vee r(Y_1, X_1)$, $\overrightarrow{C} = p(X_2, X_2) \vee r(Y_2, V_2) \vee r(U_2, Y_2)$ and $\overrightarrow{D} = p(X_3, Z_3) \vee p(Z_3, X_3) \vee r(U_3, Y_3) \vee r(Y_3, V_3)$ be ordered clauses and let $\theta_1 = \{X_2/X_1, Y_2/Y_1,\ U_2/X_1, V_2/X_1\}$. Then $\overrightarrow{C}\theta_1$ is an injective subset of $\overrightarrow{B}$ and therefore $\overrightarrow{C} \succeq^i \overrightarrow{B}$. However, there is no substitution θ_2 such that $\overrightarrow{D}\theta_2$ is an injective subset of $\overrightarrow{B}$ and therefore $\overrightarrow{D} \not\succeq^i \overrightarrow{B}$. Note that for conventional clauses B, C*

and D we have $C\theta_1 \subseteq B$ and similarly for $\theta_2 = \{X_3/X_1, Z_3/X_1, Y_3/Y_1, V_3/X_1, U_3/X_1\}$ we have $D\theta_2 \subseteq B$ and therefore $C \succeq B$ and $D \succeq B$. $\diamond$

By analogy to Definitions 53 and 55 we can define $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ and injective subsumption relative to a bottom clause.

Definition 74 **($\overrightarrow{\mathcal{L}}^{i}_{\perp}$)** *Let $\overrightarrow{\perp}$ and θ_v be as defined in Definition 53 and $\overrightarrow{C}$ a definite ordered clause. $\overrightarrow{C}$ is in $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ if $\overrightarrow{C}\theta_v$ is an injective subset of $\overrightarrow{\perp}$.*

Definition 75 (Injective subsumption relative to $\perp$) *Let $\overrightarrow{\perp}$ and $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ be as defined in Definition 74, $\theta_\perp$ be as defined in Definition 54 and $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}^{i}_{\perp}$. We say $\overrightarrow{C}$ is an injective generalisation of $\overrightarrow{D}$ relative to $\perp$, denoted by $\overrightarrow{C} \succeq^{i}_{\perp} \overrightarrow{D}$, if there exists a substitution $\theta \subseteq \theta_\perp$ such that $\overrightarrow{C}\theta$ is an injective subset of $\overrightarrow{D}$. $\overrightarrow{C}$ is a proper injective generalisation of $\overrightarrow{D}$ relative to $\perp$, denoted by $\overrightarrow{C} \succ^{i}_{\perp} \overrightarrow{D}$, if $\overrightarrow{C} \succeq^{i}_{\perp} \overrightarrow{D}$ and $\overrightarrow{D} \not\succeq^{i}_{\perp} \overrightarrow{C}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ are equivalent with respect to injective subsumption relative to $\perp$, denoted by $\overrightarrow{C} \sim^{i}_{\perp} \overrightarrow{D}$, if $\overrightarrow{C} \succeq^{i}_{\perp} \overrightarrow{D}$ and $\overrightarrow{D} \succeq^{i}_{\perp} \overrightarrow{C}$.*

Mapping function c' can be defined by analogy to Definition 68. This mapping function maps a tuple $\langle K, \theta \rangle$ into an ordered clause in $\overrightarrow{\mathcal{L}}^{i}_{\perp}$.

Definition 76 (Mapping function c') *Let $\overrightarrow{\perp}$, $\overrightarrow{\perp_v}$, K, $\mathcal{K}$, θ and Θ be as defined in Definition 68 and $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ be as defined in Definition 74. The mapping function $c' : \mathcal{K} \times \Theta \to \overrightarrow{\mathcal{L}}^{i}_{\perp}$ is defined as follows:*

$$c'(\langle K, \theta \rangle) = (\bigvee_{i \in K} l_i \text{ where } l_i \text{ is the ith literal of } \overrightarrow{\perp_v})\theta.$$

Note that in the definition of c', unlike in c, literals l_i do not need to follow the same order as literals in $\overrightarrow{\perp_v}$ and the above disjunction notion with indexes $i \in K$ means that the literals l_i of $c'(\langle K, \theta \rangle)$ can have an order different from the order of literals in $\overrightarrow{\perp_v}$.

In the following we define a refinement operator, ρ_2, which is similar to ρ_1 but uses the mapping function c' instead of c.

Definition 77 **(ρ_2)** *Let $\overrightarrow{\perp}$, $\overrightarrow{\perp_v}$, K, $\mathcal{K}$, θ, Θ and k be as defined in Definition 69 and $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ be as defined in Definition 74. Let $K \in \mathcal{K}$, $\theta \in \Theta$, $\overrightarrow{C} = c'(\langle K, \theta \rangle)$ and the mapping*

C'	θ'	K'
$\Box$	$\emptyset$	$\emptyset$
$mult(V_1,V_2,V_3) \leftarrow$	$\emptyset$	$\{1\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_4,V_5)$	$\emptyset$	$\{1,2\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5)$	$\{V_4/V_1\}$	$\{1,2\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_{12},V_{13},V_{14})$	$\{V_4/V_1\}$	$\{1,2,5\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_5,V_{13},V_{14})$	$\{V_4/V_1, V_{12}/V_5\}$	$\{1,2,5\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_5,V_2,V_{14})$	$\{V_4/V_1, V_{12}/V_5, V_{13}/V_2\}$	$\{1,2,5\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_5,V_2,V_{14}),$ $plus(V_6,V_7,V_8)$	$\{V_4/V_1, V_{12}/V_5, V_{13}/V_2\}$	$\{1,2,5,3\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_5,V_2,V_{14}),$ $plus(V_{14},V_7,V_8)$	$\{V_4/V_1, V_{12}/V_5, V_{13}/V_2,$ $V_6/V_{14}\}$	$\{1,2,5,3\}$
$mult(V_1,V_2,V_3) \leftarrow dec(V_1,V_5), mult(V_5,V_2,V_{14}),$ $plus(V_{14},V_2,V_8)$	$\{V_4/V_1, V_{12}/V_5, V_{13}/V_2,$ $V_6/V_{14}, V_7/V_2\}$	$\{1,2,5,3\}$

Figure 5.3: Example application of refinement operator ρ_2.

function c' be defined as in Definition 76. $\langle \overrightarrow{C'}, K', \theta' \rangle$ is in $\rho_2(\langle \overrightarrow{C}, K, \theta \rangle)$ if and only if $\overrightarrow{C'} = c'(\langle K', \theta' \rangle)$ and either

1. *$K' = K \cup \{k\}$, $k \notin K$ and $\theta' = \theta$ or*
2. *$K' = K$, $\theta' = \theta\{y'/x'\}$ and $\{y'/x'\} \in \Theta$ where x' and y' are distinct variables in the k_1th and k_2th literals of $\overrightarrow{\bot_v}$ respectively and k_1th and k_2th are in K'.*

The following example demonstrates how the first type of incompleteness (in Example 7) is addressed in ρ_2.

Example 34 *Let $\overrightarrow{C}$ and $\overrightarrow{\bot}$ be as defined in Example 7. Progol's refinement cannot generate C (i.e. $C \notin \rho^*(\Box)$) and also $\langle \overrightarrow{C}, K, \theta \rangle \notin \rho_1^*(\langle \Box, \emptyset, \emptyset \rangle)$. However, Figure 5.3 shows that $\langle \overrightarrow{C}, K, \theta \rangle \in \rho_2^*(\langle \Box, \emptyset, \emptyset \rangle)$, where $K = \{1,2,5,3\}$ and $\theta = \{V_4/V_1, V_{12}/V_5, V_{13}/V_2, V_6/V_{14}, V_7/V_2\}$ and $\overrightarrow{\bot_v}$ is the clause:*

$$\begin{aligned} mult(V_1,V_2,V_3) \quad \leftarrow \quad & dec(V_4,V_5), plus(V_6,V_7,V_8), plus(V_9,V_{10},V_{11}), \\ & mult(V_{12},V_{13},V_{14}), mult(V_{15},V_{16},V_{17}). \end{aligned}$$

$\diamond$

This example shows that ρ_2 can address the first type of Progol's incompleteness demonstrated in Example 7. However, ρ_2 is also more redundant than other refinement operators considered so far (e.g. ρ and ρ_1) as different permutations of the same clause could be generated in ρ_2. On the other hand, as mentioned before in this chapter, a refinement operator cannot be both complete and non-redundant.

Mode declarations can be used to consider only those permutations which are useful. For example, in Figure 5.3 the fifth literal from $\overrightarrow{\bot_v}$ is considered before the third literal because the output variables from *mult* predicate can be used as input variables for *plus*. This idea has been implemented in GA-Progol (see Chapter 7) when language $\overrightarrow{\mathcal{L}}^i_\bot$ (defined in this section) is selected. We demonstrate in Section 8.1.5 that GA-Progol can find the correct solution for special cases, such as Example 7, where the solution is beyond the exploration power of Progol's refinement operator due to its incompleteness.

Note that in the new definitions the selection functions are injective, we can therefore encode every literal of a clause by a k index from $\bot$. Hence, the properties mentioned for the mapping function c also hold for c'.

The refinement operator ρ_2 defined in this section is similar to the refinement operator $\rho^{(1)}_\bot$ introduced in [BS99]. However, the subsumption order used in [BS99] (i.e. weak subsumption) is a special case of the subsumption order introduced in this section. Using different conditions on the selection functions in Definition 72, we can get different kind of subsumption orders. For example, if the selection function is monotonically increasing then we will have a subsumption order which allows each literal of $\bot$ to be selected more than once. In this case, Definition 72 will be similar to the definition of subsequences considered in [KOHH06]. This will address the second type of Progol's incompleteness mentioned before. However, the selection functions are not injective and therefore the encoding and the morphism we described in this paper are not applicable.

In this chapter we only studied downward refinement operators for bounded subsumption. In a downward refinement operator relative to $\bot$ (e.g. ρ_1 and ρ_2), a clause is specialised by adding a literal or by unifying variables. Similarly, an upward refinement operator relative to $\bot$ can be defined by removing a literal or splitting variables. In fact, every refinement operator relative to $\bot$ can be defined based on basic refinement operators relative to $\bot$ as defined in the following.

Definition 78 (Basic refinement operators $\rho^a_\bot$, $\rho^d_\bot$, $\rho^u_\bot$ and $\rho^s_\bot$) *Let $\overrightarrow{\bot}$ and $\overrightarrow{\mathcal{L}}_\bot$ be as defined in Definition 53, $\overrightarrow{C}$ be an ordered clause in $\overrightarrow{\mathcal{L}}_\bot$, n be the number of* *literals in $\bot$, k be a natural number, $1 \leq k \leq n$, $\overrightarrow{\bot_v}$, Θ and $\mathcal{K}$ be defined as in Definition 68. Let $K \in \mathcal{K}$, $\theta \in \Theta$, $\overrightarrow{C} = c(\langle K, \theta \rangle)$ and the mapping function c be defined as in Definition 68.*

Add literal ($\rho^a_\bot$): *$\langle \overrightarrow{C'}, K', \theta' \rangle$ is in $\rho^a_\bot(\langle \overrightarrow{C}, K, \theta \rangle)$ if and only if $\overrightarrow{C'} = c(\langle K', \theta' \rangle)$ and $K' = K \cup \{k\}$, $k \notin K$ and $\theta' = \theta$.*

Delete literal ($\rho^d_\bot$): *$\langle \overrightarrow{C'}, K', \theta' \rangle$ is in $\rho^d_\bot(\langle \overrightarrow{C}, K, \theta \rangle)$ if and only if $\overrightarrow{C'} = c(\langle K', \theta' \rangle)$ and $K = K' \cup \{k\}$, $k \notin K$ and $\theta' = \theta$.*

Unify variables ($\rho^u_\bot$): *$\langle \overrightarrow{C'}, K', \theta' \rangle$ is in $\rho^u_\bot(\langle \overrightarrow{C}, K, \theta \rangle)$ if and only if $\overrightarrow{C'} = c(\langle K', \theta' \rangle)$ and $K' = K$ and $\theta' = \theta\{y'/x'\}$ and $\{y'/x'\} \in \Theta$ where x' and y' are distinct variables in the k_1th and k_2th literals of $\overrightarrow{\bot_v}$ respectively and k_1th and k_2th are in K'.*

Split variables ($\rho^s_\bot$): *$\langle \overrightarrow{C'}, K', \theta' \rangle$ is in $\rho^s_\bot(\langle \overrightarrow{C}, K, \theta \rangle)$ if and only if $\overrightarrow{C'} = c(\langle K', \theta' \rangle)$ and $K' = K$ and $\theta = \theta'\{y'/x'\}$ and $\{y'/x'\} \in \Theta$ where x' and y' are distinct variables in the k_1th and k_2th literals of $\overrightarrow{\bot_v}$ respectively and k_1th and k_2th are in K'.*

An ILP system can be based on one or more of the basic refinement operators described in Definition 78. For example the Asymmetric Relative Minimal Generalisation (ARMG) operator implemented in ProGolem (i.e $armg_\bot$) is based on removing literals from $\bot$. GA-Progol implements all of the basic refinement operators described in this definition. ARMGs, ProGolem and GA-Progol are discussed in Chapter 7.

5.5 Related work and discussion

In this chapter we used an encoding of clauses with respect to a bottom clause. In this encoding each clause is represented by a tuple $\langle K, \theta \rangle$ and it can be constructed from $\overrightarrow{\bot_v}$ as described in Definition 68. This idea was first used in [TNM00] where the substitution θ is encoded as a binding matrix which maps the variables of $\overrightarrow{\bot_v}$ to the variables of a clause with respect to the bottom clause. The morphism between the lattice of variable bindings and the subsumption lattice was also studied in [TNM00]. The encoding and refinement operators for the bounded subsumption are also related to stochastic searches in ILP. For example, the refinement operator ρ_1 which is ideal is the basis of some stochastic refinement operators which are discussed in Chapter 7. In a systematic and structured search, such as the A^*-like search in Progol, the search always starts from the empty clause ($\Box$) and a weakly complete refinement operator is sufficient to generate any clauses in the language. However, in a stochastic search which starts from random points in the search space, we need a complete refinement operator rather than a weakly complete operator. This is because for each clause C and D in $\mathcal{L}$ if we have $C \succeq D$ and the search is started from C then D should be accessible from C. In other words, if $C \succeq D$ then there should always be a ρ-chain from C to D and therefore ρ should be a complete (downward) refinement operator. Hence, from the two favourable sets of properties for refinement operators, i.e. idealness and optimality, the first one is more appropriate for a stochastic strategy. On the other hand, it is known that a refinement operator can not be both complete and non-redundant. A complete refinement operator is therefore redundant. The refinement operator ρ_1 described in this chapter works on an encoding of a clause, i.e. the encoding tuples $\langle K, \theta \rangle$, rather than the clause itself. This property is also used in the stochastic refinement operators discussed in Chapter 7.

The $lgg_\perp$ (and $armg_\perp$) operators discussed in this chapter can be compared with other approaches which use lgg-like operators but instead of considering all pairs of compatible literals they only consider one pair. For example, LOGAN-H [AK04] is a bottom-up system which is based on inner products of examples which are closely related to the lgg operator. This system constructs lgg-like clauses by considering only those pairs of literals which guarantee an injective mapping between variables. In other words, it assumes one-to-one object mappings. Other similar approaches use the same idea of simplifying the lgg-like operations by considering only one pair of compatible literals but they select this pair arbitrarily (e.g. [BZB01]).

As in [BS99], the refinement operator ρ_1 defined in this chapter is based on Laird's operator [Lai87] adapted for subsumption relative to $\perp$. However, as discussed in Chapter 3 the approach in [BS99] is based on weak subsumption which is different from the subsumption relative to $\perp$ (e.g. it cannot capture the ordering of the literals in Progol's refinement). The refinement operator ρ_2 is also similar to the refinement operator $\rho_\perp^{(1)}$ introduced in [BS99]. But again the subsumption order used in [BS99] (i.e. weak subsumption) is different and also no completeness proof was given in [BS99].

In this chapter we have studied alternative forms of subsumption relative to $\perp$. Using different conditions on the selection function in Definition 72, we can get different kind of subsumption orders relative to $\perp$. For example, if the selection function is monotonically increasing then we will have a subsumption order which allows each literal of $\perp$ to be selected more than once. In this case, Definition 72 will be similar to the definition of subsequences considered in [KOHH06]. This will address the second type of Progol's incompleteness discussed in Chapter 3, i.e. incompleteness due to the fact that each literal from the bottom clause can be only selected once. However, the selection functions are not injective and therefore the encoding and the morphism described in this chapter are not applicable.

In this chapter we have shown that ρ_1 is ideal for subsumption relative to $\perp$ and as discussed above the idealness property is more appropriate than optimality for stochastic searches which we are interested in this thesis. However, in other contexts especially when using a systematic search, an optimal operator could be more appropriate. It is shown (e.g. [BS99]) that for each ideal refinement operator ρ we can construct an optimal refinement operator $\rho^{(o)}$. Hence, we can construct a refinement operator $\rho_1^{(o)}$ which is optimal for subsumption relative to $\perp$ from the ideal refinement operator ρ_1. The procedure for this is as follows. $\rho_1^{(o)}$ is obtained from ρ_1 such that for $D \in \rho_1(C_1) \cap \cdots \cap \rho_1(C_n)$ we have $\exists i.D \in \rho_1^{(o)}(C_i)$ and $\forall j \neq i.D \notin \rho_1^{(o)}(C_j)$.

5.6 Summary

In this chapter we have defined a refinement operator ρ_1 for the lattice of bounded subsumption, i.e. $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. We have shown that ρ_1 is ideal and therefore, unlike for the general subsumption order, ideal refinement operators exist for the lattice of bounded subsumption. Each clause $\overrightarrow{C}$ in this lattice is encoded by a tuple $\langle K, \theta \rangle$, where K is a set of indexes from the bottom clause and θ is a variable subsumption which maps variables between $\overrightarrow{C}$ and $\overrightarrow{\perp}_v$. The refinement operator ρ_1 works on the encoding tuples and the mapping function c, maps a tuple $\langle K, \theta \rangle$ into an ordered clause $\overrightarrow{C}$ in $\overrightarrow{\mathcal{L}}_\perp$. This refinement operator is the basis of the genetic operators which work on the encoding of the clauses in a GA-ILP search (see Chapter 7). In this chapter we also studied the mapping and the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and a lattice of the encoding tuples $\langle K, \theta \rangle$. We have shown that the mapping function c is a lattice isomorphism and lattices $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \subseteq \rangle$ are two isomorphic lattices. In this chapter we have also studied alternative forms of subsumption relative to $\perp$. We have shown that using different conditions on the selection function which maps the literals from a clause to the literals of $\perp$, we can get different forms of subsumption orders relative to $\perp$. We have defined the refinement operator ρ_2 based on an alternative form of bounded subsumption and we demonstrated how ρ_2 can address the first type of Progol's incompleteness (i.e. incompleteness due to the ordering of the literals).

Chapter 6

Stochastic refinement and genetic search

Most ILP systems are traditionally based on clause refinement through a lattice defined by a generality order (e.g. subsumption) as discussed in Chapter 2. However, there is a long-standing and increasing interest in stochastic search methods in ILP for searching the space of candidate clauses (e.g. [PKK93, Sri00, TNM02, RK03, ZSP06, PŽZ^{+}07, MTN07, DPZ08]). The idea of stochastic refinement presented in this chapter is motivated by the following question. How can the generality order of clauses and the relevant concepts such as refinement be adapted to be used in a stochastic search? To address this question we introduce the concept of stochastic refinement operators and adapt a framework, called stochastic refinement search. As shown in Chapter 2, refinement of a clause is defined as a set of clauses. In this chapter we introduce stochastic refinement of a clause as a probability distribution over a set of clauses. This probability distribution can be viewed as a Bayesian prior over the hypotheses in a stochastic ILP search. We also define the concept of stochastic refinement search. In general a stochastic refinement search can be viewed as a Markov chain in which the next state of the search only depends on the current state. We study the properties of a stochastic refinement search as two well known Markovian approaches: 1) Gibbs sampling algorithm [Mit97] and 2) random heuristic search [Vos99]. As a Gibbs sampling algorithm, a stochastic refinement search iteratively generates random samples from the hypothesis space according to a posterior distribution. We define a special case of random heuristic search called monotonic random heuristic search and we show that due to the generality order defined by the refinement operators, a stochastic refinement search can be viewed as a monotonic random heuristic search.

In the second part of this chapter we discuss Genetic Algorithms (GAs) as search methods for learning first-order clauses. We discuss several learning systems which use genetic and evolutionary approaches for learning first-order clauses. We show that these systems use a limited form of background knowledge (e.g. to seed the population) and they cannot benefit from intentional background knowledge in the same way as in an ILP system. In this chapter we discuss a hybrid GA-ILP framework in which background knowledge can be used in the same way as in the ILP systems such as Progol and Aleph. This GA-ILP framework not only uses a standard ILP representation but we also show that the proposed encoding is isomorphic to the bounded subsumption lattice. This property can be used to design task-specific genetic operators as in GA-Progol. The framework of stochastic refinement relative to a bottom clause described in this chapter is the basis of the algorithms (e.g. GA-Progol, QG, QG/GA and ProGolem) which are discussed in Chapter 7. The concept of stochastic refinement operators is also used in the ILP system MetaBayes [MLCTN13] which is based on higher-order stochastic refinement.

This chapter is organised as follows. In Section 6.1 stochastic refinement operators are introduced and their properties are discussed. The framework of stochastic refinement search is discussed in Section 6.2 and this framework is used to characterise stochastic search methods in some ILP systems. In Section 6.3 we discuss genetic search for learning first-order clauses. In Section 6.4 we describe a framework for genetic and stochastic refinement search for bounded subsumption. Related work is discussed in Section 6.5 and Section 6.6 summarises the chapter.

6.1 Stochastic refinement operators

In this section we introduce the concept of stochastic refinement operators. According to Definition 36 (and Definition 37), refinement of a clause (or a pair of clauses) is a set of clauses. In the following we define stochastic refinement as a probability distribution over a set of clauses. In this setting each clause in the refinement space is associated with a probability.

Definition 79 (Stochastic unary refinement operator) *Let ρ be a (downward) unary refinement operator for the quasi-ordered set $\langle G, \succeq \rangle$ and $C \in G$. A (downward)* stochastic unary refinement operator *is a function $\sigma : G \rightarrow 2^{G \times [0,1]}$ defined as follows: $\sigma(C) = \{\langle D_i, p_i \rangle | D_i \in \rho(C),\ p_i \in [0,1]$ and $\sum p_i = 1$ for $1 \leq i \leq |\rho(C)|\}$.*

- *A σ-**chain** from C to D, denoted by $C \xrightarrow{\sigma} D$, is a sequence $C = C_0, C_1, \ldots, C_m = D$, such that $\langle C_i, p_i \rangle \in \sigma(C_{i-1})$ for every $1 \leq i \leq m$ and the probability of this* σ-chain *is $p(C \xrightarrow{\sigma} D) = \prod_{i=1}^{m} p_i$.*
- ***n-step stochastic refinements** of C is defined as: $\sigma^n(C) = \{\langle D, p \rangle | D \in \rho^n(C)$ and $p = \sum_{x \in X} p(x)$ where X is the set of σ-chains from C to $D\}$.*
- ***stochastic refinements** of C is defined as: $\sigma^*(C) = \{\langle D_i, p_i \rangle | D_i \in \rho^*(C),\ p_i \in [0,1]$ and $\sum p_i = 1$ for $1 \leq i \leq |\rho^*(C)|\}$.*
- *σ inherits the standard properties (i.e. local finiteness, properness and completeness) from ρ.*
- *σ is* ε-complete *if for every $C, D \in G$ such that $C \succ D$, there is an $\langle E, p \rangle \in \sigma^*(C)$ such that $D \sim E$ (i.e. D and E are equivalent in the $\succeq$-order) and $p > \varepsilon$.*
- *σ is* weakly ε-complete *for $\langle G, \succeq \rangle$ if for each $C \in G$ there is a $\langle C, p \rangle \in \sigma^*(\Box)$, where $p > \varepsilon$ and $\Box$ is the top element of G.*

Example 35 *Figure 6.1.a shows refinement of a clause as a set of clauses. Stochastic refinement of a clause is defined as a probability distribution over a set of clauses (Figure 6.1.b).*

Example 36 *Figure 6.2 shows part of a stochastic refinement graph. This graph shows the probabilities of clauses in $\sigma^n(C)$ as defined in Definition 79. For example, there are two σ-chains from C to D_4. Hence, the probability of D_4 in $\sigma^2(C)$ is $0.5 \times 1.0 + 0.3 \times 0.4 = 0.62$.*

According to Definition 79, $\sigma(C)$ and $\sigma^*(C)$ represent probability distributions. In the following we show that $\sigma^n(C)$ is also a probability distribution.

Theorem 19 *n-step stochastic refinements of a clause represent a probability distribution.*

$$\begin{tikzcd}
C: & & p(x,y) \arrow[dl] \arrow[d] \arrow[dr] & \\
\rho(C): & p(x,x) & p(x,y)\leftarrow q(x,z) & p(x,y)\leftarrow r(w,y)
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
C: & & p(x,y) \arrow[dl, "0.5"'] \arrow[d, "0.3"] \arrow[dr, "0.2"] & \\
\sigma(C): & p(x,x) & p(x,y)\leftarrow q(x,z) & p(x,y)\leftarrow r(w,y)
\end{tikzcd}$$

(b)

Figure 6.1: (a) Refinement of a clause is defined as a set of clauses (b) Stochastic refinement of a clause is defined as a probability distribution over a set of clauses.

$$\begin{tikzcd}
C: & & C \arrow[dl, "0.5"'] \arrow[d, "0.3"] \arrow[dr, "0.2"] & & \\
\sigma(C): & \langle D_1, 0.5\rangle \arrow[d, "1.0"] & \langle D_2, 0.3\rangle \arrow[dl, "0.4"'] \arrow[d, "0.4"] \arrow[dr, "0.2"] & \langle D_3, 0.2\rangle \arrow[dr, "1.0"] & \\
\sigma^2(C): & \langle D_4, 0.62\rangle \arrow[d] & \langle D_5, 0.12\rangle \arrow[d] & \langle D_6, 0.06\rangle \arrow[d] & \langle D_7, 0.2\rangle \arrow[d] \\
\sigma^n(C): & \dots & \dots & \dots & \dots
\end{tikzcd}$$

Figure 6.2: Part of a stochastic refinement graph.

Proof. Let $\sigma^n(C)$ be n-step stochastic refinements of clause C as defined in Definition 79 such that $\sigma^n(C) = \{\langle D_1, p_1\rangle, \langle D_2, p_2\rangle, \dots \langle D_m, p_m\rangle\}$. We will show that $\sum_{i=1}^{m} p_i = 1$. The proof is by induction on n. For $n = 1$ the theorem is true by definition of $\sigma(C)$. Assume that the theorem is true for k then the sum of probabilities $p_1, p_2, \dots, p_s$ for s nodes at level k is 1: $\sum_{i=1}^{s} p_i = 1$. Suppose that each node with probability p_i at level k is extended into t_i nodes with probabilities $q_{i1}, q_{i2}, \dots, q_{it_i}$. Then the sum of probabilities at level $k+1$ will be: $\sum_{i=1}^{s}\sum_{j=1}^{t_i} p_i q_{ij} = \sum_{i=1}^{s} p_i(q_{i1} + q_{i2} + \dots + q_{it_i})$. But $q_{i1} + q_{i2} + \dots + q_{it_i} = 1$ and $\sum_{i=1}^{s} p_i = 1$ and therefore the sum of probabilities at level $k+1$ will be 1 and this completes the proof. $\square$

Example 37 *In Figure 6.2, $\sigma(C)$ represents a probability distribution and the sum of probabilities for clauses in $\sigma(C)$ is equal to 1. According to Theorem 19, $\sigma^n(C)$ is also a probability distribution and as shown in this figure the sum of probabilities for clauses in $\sigma^2(C)$ is equal to 1.*

In the following we define analogous concepts for stochastic binary refinement operators.

Definition 80 (Stochastic binary refinement operator) *Let ρ be a (downward) binary refinement operator for the quasi-ordered set $\langle G, \succeq \rangle$ and $C, D \in G$. A (downward)* stochastic binary refinement operator *is a function $\sigma : G^2 \rightarrow 2^{G\times[0,1]}$ defined as: $\sigma(C,D) = \{\langle E_i, p_i\rangle | E_i \in \rho(C,D),\ p_i \in [0,1]$ and $\sum p_i = 1$ for $1 \leq i \leq |\rho(C,D)|\}$.*

- *A **σ-chain** from (C,D) to E, denoted by $(C,D) \xrightarrow{\sigma} E$, is a sequence $(C,D) = (C_0,D_0),(C_1,D_1),\dots,(C_m,D_m)$, such that $E = C_m$ or $E = D_m$ and $\langle C_i,p_i\rangle, \langle D_i,q_i\rangle \in \sigma(C_{i-1},D_{i-1})$ for every $1 \leq i \leq m$ and the probability of this* σ-chain *is $p((C,D) \xrightarrow{\sigma} E) = \prod_{i=1}^{m} p_i q_i$.*
- ***n-step stochastic binary refinements** of some $(C,D) \in G^2$ is defined as: $\sigma^n(C,D) = \{\langle E,p\rangle | E \in \rho^n(C,D)$ and $p = \sum_{x\in X} p(x)$ where X is the set of σ-chains from (C,D) to $E\}$.*
- ***stochastic binary refinements** of some $(C,D) \in G^2$ is defined as: $\sigma^*(C,D) = \{\langle E_i,p_i\rangle | E_i \in \rho^*(C,D),\ p_i \in [0,1]$ and $\sum p_i = 1$ for $1 \leq i \leq |\rho^*(C,D)|\}$.*
- *σ inherits the standard properties (i.e. local finiteness, properness and completeness) from ρ.*
- *σ is* ε-complete *if for every $B, C, D \in G$ such that $C \succ B$, $D \succ B$, there is an $\langle E,p\rangle \in \sigma^*(C,D)$ such that $B \sim E$ (i.e. D and E are equivalent in the $\succeq$-order) and $p > \varepsilon$.*
- *σ is* weakly ε-complete *for $\langle G, \succeq\rangle$ if for each $C \in G$ there is a $\langle C,p\rangle \in \sigma^*(\Box,\Box)$, where $p > \varepsilon$ and $\Box$ is the top element of G.*

Example 38 *Figure 6.3 shows stochastic binary refinement of a pair of clauses as a probability distribution over a set of clauses.*

As for $\sigma^n(C)$, it can be shown that $\sigma^n(C,D)$ is a probability distribution.

Theorem 20 *n-step stochastic binary refinements of a pair of clauses represent a probability distribution.*

Proof. Let $\sigma^n(C,D)$ be n-step stochastic binary refinements of clause C and D as defined in Definition 80 such that $\sigma^n(C,D) = \{\langle E_1,p_1\rangle, \langle E_2,p_2\rangle, \dots \langle E_m,p_m\rangle\}$. We

$$\begin{tikzcd}
(C,D): & & C \arrow[dl] \arrow[dr] \arrow[drrr] & & D \arrow[dlll] \arrow[dl] \arrow[dr] & \\
\sigma(C,D): & \langle E_1, 0.4\rangle & & \langle E_2, 0.4\rangle & & \langle E_3, 0.2\rangle
\end{tikzcd}$$

Figure 6.3: Stochastic binary refinement of a pair of clauses is defined as a probability distribution over a set of clauses.

will show that $\sum_{i=1}^{m} p_i = 1$. The proof is by induction on n and similar to the proof of Theorem 19. For $n = 1$ the theorem is true by definition of $\sigma^n(C, D)$. Assume that the theorem is true for k then the sum of probabilities $p_1, p_2, \ldots, p_s$ for s nodes at level k is 1: $\sum_{i=1}^{s} p_i = 1$. Suppose that each pair of nodes with probabilities p_i and p_j at level k is extended into t_{ij} nodes with probabilities $r_{ij1}, r_{ij2}, \ldots, r_{ijt_{ij}}$. Then the sum of probabilities at level $k+1$ will be: $\sum_{i=1}^{s} \sum_{j=1}^{s} \sum_{l=1}^{t_{ij}} p_i q_j r_{ijl} = \sum_{i=1}^{s} \sum_{j=1}^{s} p_i q_j (r_{ij1} + r_{ij2} + \cdots + r_{ijt_{ij}})$. But $r_{ij1} + r_{ij2} + \cdots + r_{ijt_{ij}} = 1$ and $\sum_{i=1}^{s} \sum_{j=1}^{s} p_i q_j = \sum_{i=1}^{s} p_i (q_1, q_2, \ldots, q_s) = \sum_{i=1}^{s} p_i = 1$ and therefore the sum of probabilities at level $k + 1$ will be 1 and this completes the proof. $\square$

6.2 Stochastic refinement search

Different stochastic search methods have been used to explore the space of candidate clauses in ILP. Examples of these search methods are simulated annealing (e.g.[PKK93, SPR04a]), genetic algorithms (e.g. [TNM02]) and randomised restarted search (e.g. [ZSP06]). In this section we define a general framework which can be used to characterise some of the existing stochastic search methods in ILP and can also be used as a basis to design new stochastic methods .

Definition 81 (Stochastic refinement search) *Let G and σ be defined as in Definition 79 and S_0 be a sample from G with size s. A stochastic refinement search involves a sequence $S_0 \xrightarrow{\sigma} S_1 \xrightarrow{\sigma} S_2 \xrightarrow{\sigma} \ldots$ where S_{i+1} is generated from S_i such that for each $C_{i+1} \in S_{i+1}$ there exists $C_i \in S_i$ such that $\langle C_{i+1}, p\rangle \in \sigma(C_i)$. Similarly a stochastic refinement search can be defined for a stochastic binary refinement operator σ (Definition 80) where for each $C_{i+1} \in S_{i+1}$ there exist C_i and $D_i \in S_i$ such that $\langle C_{i+1}, p\rangle \in \sigma(C_i, D_i)$.*

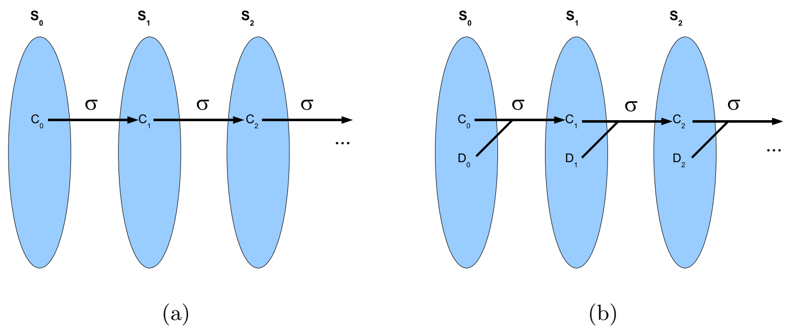


Figure 6.4: A stochastic refinement search with (a) unary stochastic refinement operator (b) binary stochastic refinement operator.

Figure 6.4 shows stochastic refinement search with unary and binary stochastic refinement operators. In Definition 81, the initial sample S_0 corresponds to the starting point of the search as in simulated annealing or an initial population as in genetic algorithms. These are usually generated randomly. The sample size s is equal to one in simulated annealing and it is the population size in a genetic algorithm. Stochastic refinement operator σ corresponds to task-specific operators or search transition rules in an stochastic ILP search algorithm. These operators generate new clauses from the clauses in the previous search state or population. For example, in the simulated annealing search used in ILP (e.g.[PKK93, SPR04a]), the transition operators can be viewed as downward stochastic unary refinement operators which stochastically choose the next literal to be added to the body of a given clause. Crossover operator $lgg_\perp$ of the genetic algorithm search used in ILP (e.g. [TNM02]) can be viewed as a stochastic binary refinement operator. Note that some stochastic refinement searches (e.g. genetic algorithms) use both stochastic unary and binary refinement operators.

According to Definition 81 a stochastic refinement search is a sequence of random samples with the property that the current state depends only on the previous state. Hence, in general a stochastic refinement search can be viewed as a Markov chain. In the following sections we study the properties of a stochastic refinement search as two well known Markovian approaches: 1) a Gibbs sampling algorithm and 2) a random heuristic search.

6.2.1 Stochastic refinement search as a Gibbs sampling algorithm

According to [Mit97], many machine learning algorithms, whether they explicitly manipulate probabilities or not, can be viewed as approximations to the following general Bayesian inference approaches: i) methods which find a Maximum A Posterior (MAP) hypothesis; ii) Bayes optimal classifier and iii) Gibbs classifier. In the following we briefly describe these approaches and show that in general a stochastic refinement search can be viewed as a Gibbs sampling algorithm.

According to Bayes' theorem, the posterior probability of hypothesis H given examples E can be defined as follows:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

where $P(H)$ and $P(E)$ are prior probabilities of H and E respectively and the conditional probability $P(E|H)$, also known as the likelihood, is 1 when H explains E and 0 otherwise. MAP-based algorithms return a hypothesis with maximum posterior probability, i.e. $H_{MAP} = argmax_H P(H|E)$. Some algorithms (including many ILP systems) are based on the Minimum Description Length (MDL) principle [Ris78] which, under some assumptions, can approximate a MAP hypothesis [Mit97]. In ILP systems such as Progol, this is approximated by finding the hypothesis which provides the highest (positive) *compression* over examples [Mug95]. Here, *compression* is an information-theoretic criterion which can be defined as $compression = p - n - h$, where p is the number of observations correctly explained by the hypothesis, n is the number incorrectly explained and h is the length of the hypothesis. Instead of finding a MAP hypothesis, a Bayes optimal classifier classifies unseen instances based on weighted joint prediction of the entire hypothesis space. Bayes optimal classifier provides maximum overall predictive accuracy [Mit97], but it can be expensive as it may involve many hypotheses. There is however a less expensive algorithm called Gibbs classifier which can approximate Bayes optimal classifier. A Gibbs classifier chooses one hypothesis at random according to the posterior probability $P(H|E)$ and uses this to classify a new instance. A Gibbs classifier can be used to randomly sample hypotheses according to the posterior distribution. It has been shown [HKS94] that the expected error for Gibbs algorithm is at most twice the expected error for Bayes optimal classifier.

All of the above algorithms assume a Bayes' prior distribution over the hypothesis space and in the case of Gibbs this is used for sampling. It has been shown (e.g. [GCSR03]) that the sequence of samples in a Gibbs sampling algorithm constitutes a Markov chain. Stochastic refinement search introduced in this chapter assumes a prior distribution over the hypothesis space which is defined by stochastic refinement operators. Hence, a stochastic refinement search can be viewed as a Gibbs sampling algorithm in which each new hypothesis is selected randomly according to a posterior probability. As noted in [MTN07], some stochastic machine learning algorithms can be viewed as Gibbs-MAP algorithms. A Gibbs-MAP algorithm is a Gibbs-like approximation to MAP based on sampling. A stochastic refinement search can also be turned into Gibbs-Map which aims at maximising the posterior probability by iteratively generating new samples from the hypothesis space.

The Quick Generalisation (QG) algorithm described in [MTN07] is a Gibbs-MAP algorithm, which by construction, generates consistent clauses by stochastically pruning Progol bottom clauses. The QG sampling algorithm can also be viewed as a stochastic refinement search with unary refinement operator which randomly samples from "fringe" clauses (i.e. maximally general consistent clauses in the hypothesis space). A sampling algorithm based on QG is described in Chapter 7. This sampling algorithm returns the clause with highest positive compression from a sample of s calls to QG. In Chapter 7, we also describe an algorithm based on Asymmetric Relative Minimal Generalisation (ARMG) [MSTN10a] which can be viewed as a stochastic refinement search. In this section we show how a proper sample size can be selected to guarantee that with high probability a consistent and compressive hypothesis is generated by a stochastic refinement search.

In the QG sampling algorithm mentioned above, the sample size s is set by the user and the algorithm simply returns a consistent clause with the highest positive compression. There is a trade-off between the efficiency and the achieved compression which can be controlled by the sample size. The algorithm can be made arbitrarily efficient by choice of sample size (down to 1). However, a good sample size should guarantee that with a high probability a consistent and compressive hypothesis can be generated.

The following Proposition shows that a minimum sample size can be estimated based on the percentage of consistent clauses which are compressive (i.e. have positive compression).

Proposition 21 *Suppose that an algorithm randomly and independently samples from consistent clauses and the probability that a clause does not have a positive compression is* p*. Suppose that we randomly sample* s *clauses in each iteration. Then the probability that there is at least one compressive clause in* s *samples is* $1 - p^s$*.*

Proof. The probability that there is no compressive clause in s samples is p^s. Hence, the probability that there is at least one compressive clause in s samples is $1 - p^s$.

Example 39 *The probability that a randomly generated consistent clause does not have positive compression can be estimated by the proportion of consistent clauses which are not compressive. Suppose that 1 out of 5 consistent clauses have positive compression, then according to Proposition 21 the probability of finding a consistent and compressive clause can be estimated by* $1 - (4/5)^s$ *where* s *is the sample size. Then, by choosing* $s = 10$ *this probability will be around 0.9.*

As mentioned above, Progol can be viewed as a MAP-based algorithm, which uses compression criterion to approximate a MAP hypothesis, and QG as a Gibbs-MAP algorithm. Unlike QG, Progol cannot be viewed as stochastic refinement search, because it uses a A^*-like search and maintains a growing search graph rather than a set of samples. However, Golem [MF90] can be viewed as as stochastic refinement search which employs determinate least general generalisation under relative subsumption (RLGG). Golem combines random sampling and a hill-climbing search to construct RLGGs with randomly sampled positive examples at each iteration. The refinement in Golem can be viewed as an upward *lgg*-like stochastic refinement. Golem maintains a set of RLGGs which are further generalised with new sampled examples at each iteration until it converges to a set of consistent clauses each covering a partition of positive examples.

Unlike QG, Golem's stochastic refinement does not guarantee to generate consistent clauses. However, the following theorem shows how the sample size and the probability of generating a consistent clause are related. This can be used to set a minimum sample size such that with a high probability at least one consistent RLGG is generated.

Theorem 21 *Let k be an upper bound on the number of clauses in a consistent target theory, c be a natural number and E^+ and E^- be the set of positive and negative examples respectively. Suppose that $s = ck$ pairs of clauses are randomly sampled from E^+. Then the probability that there is a pair of clauses C_1 and C_2, such that $lgg(C_1, C_2)$ is consistent with E^-, is at least $1 - e^{-c}$.*

Proof. First suppose that the k clauses in the target theory have disjoint coverage and each covers the same number of clauses. In this case, there are k partitions each covered by one clause in the target theory. The probability that a randomly selected pair of clauses belong to the same partition is $\frac{1}{k}$. If we randomly sample $s = ck$ pairs of clauses, the probability that there is no pair of clauses belonging to the same partition is $(1 - \frac{1}{k})^{ck}$. If the coverages are of different sizes then the probability that there is not a pair of clauses covered by the same clause in the target theory is less than $(1 - \frac{1}{k})^{ck}$. This is because the product of the probabilities are maximum if the coverages have the same size. Similarly, if the coverages are not disjoint then the probability that there is not a pair of clauses covered by the same clause in the target theory is less than $(1-\frac{1}{k})^{ck}$. Also note that the maximum value for $(1 - \frac{1}{k})^{ck}$ is $\lim_{k\to\infty}(1 - \frac{1}{k})^{ck} = e^{-c}$. Then the probability that a randomly selected pair of clauses C_1 and C_2 are both covered by the same target clause is at least $1 - e^{-c}$. But if C_1 and C_2 are both covered by the same target clause C then $lgg(C_1, C_2)$ is consistent because by definition $lgg(C_1, C_2)$ is more specific than C which is consistent. Hence, the probability that there is a pair of clauses C_1 and C_2 such that $lgg(C_1, C_2)$ is consistent is at least $1 - e^{-c}$. $\square$

Example 40 *Consider a stochastic refinement search algorithm such as Golem which uses lgg-like operators. Suppose that the upper bound on the number of clauses in the target theory is k. Then according to Theorem 21, by selecting a minimum sample size $s = 2k$, the probability that a consistent lgg is generated is at least $1 - e^{-2} = 0.86$. With a sample size of $s = 3k$, the probability is increased to $1 - e^{-3} = 0.95$*

6.2.2 Stochastic refinement search as a random heuristic search

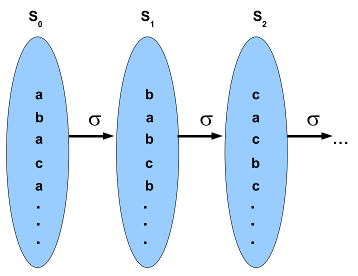
It has been shown [Vos99] that many stochastic search methods including simulated annealing, stochastic beam search and genetic algorithms are instances of a general framework called random heuristic search. The basic conditions are that the search space (Ω) be finite and that the search transition rules (τ) be Markovian and expressible as the result of s independent identically distributed random choices. The finiteness condition is usually met in most cases, including ILP, since in practice it is common to use a depth bound which leads to a finite search space. In this section we show that a stochastic refinement search is a special case of random heuristic search. First we describe a random heuristic search. The following definition is adapted from [Vos99].

Definition 82 (Random heuristic search) *Let $\Omega = \{x_0, x_1, \ldots, x_{n-1}\}$ be a search space and P_0 be a sample from Ω with size s. A random heuristic search involves a sequence $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$ where each sample P_{i+1} is generated from previous sample P_i. Each sample P_i can be represented by a probability vector $p_i = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ such that t_j is the proportion of x_j in P_i. Hence, a random heuristic search $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$ can also be denoted as a sequence of corresponding probability distributions $p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \ldots$ where p_i is the probability vector for P_i and this sequence is generated by iterating a transition rule $\tau : \Delta^n \rightarrow \Delta^n$ where $\Delta^n = \{\langle s_0, s_1, \ldots, s_{n-1}\rangle | \sum_{j=0}^{n-1} s_j = 1,\ s_j \geq 0$ for all $s_j\}$.*

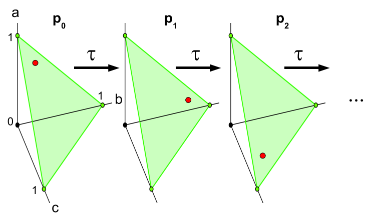
Example 41 *Suppose in Definition 82 we have $\Omega = \{0, 1, 2, 3, 4, 5\}$ and $P_0 = \{1, 0, 3, 1, 1, 3, 2, 2, 4, 0\}$. Then, P_0 can be represented by the probability vector $p_0 = \langle 0.2, 0.3, 0.2, 0.2, 0.1, 0.0\rangle$. Here the proportional representation given by p_0 determines the sample P_0 once the sample size is known.*

Note that in Definition 82, Δ^n serves as both the space of samples of Ω and the space of probability distributions over Ω.

It follows from Definitions 81 and 82 that a stochastic refinement search is an instance of random heuristic search. Figure 6.5 compares stochastic refinement search versus random heuristic search. As shown in this figure, a stochastic refinement operator σ operates on the elements of a sample S_i while a transition rule τ works directly



(a)



(b)

Figure 6.5: (a) Stochastic refinement search (b) Random heuristic search. A stochastic refinement operator σ, operates on the elements of a sample S_i while a transition rule τ works directly on the corresponding probability vector p_i.

on the corresponding sample distribution p_i. A main difference between a stochastic refinement search and a random heuristic search is that in general, a random heuristic search does not consider any ordering over Ω or for the transition rule τ. However, a stochastic refinement search is a directed search due to the generality order defined by the refinement operators. In the following we define a monotonic random heuristic search and show that an upward (or downward) stochastic refinement search can be viewed as a monotonic random heuristic search.

Definition 83 (Monotonic random heuristic search) *Let the search space Ω and the random heuristic search $P_0 \to P_1 \to P_2 \to \dots, \dots$ be defined as in Definition 82 and $\leq$ be a binary relation on Ω such that $\langle \Omega, \leq \rangle$ is a quasi-ordered set. If for each i, $x \in P_i$ is replaced by $x' \in P_{i+1}$ and we have $x \leq x'$ then the random heuristic search $P_0 \to P_1 \to P_2 \to \dots$ is said to be monotonic with respect to $\leq$.*

The following Proposition follows directly from Definitions 36, 81 and 83.

Proposition 22 *A stochastic refinement search is a monotonic random heuristic search.*

Proof. According to Definitions 36 and 81, for each i, $C_i \in S_i$ and $C_{i+1} \in S_{i+1}$ we have $C_i \succeq C_{i+1}$ and therefore according to Definition 83 a stochastic refinement search is a monotonic random heuristic search with respect to $\succeq$.

The advantage of studying stochastic refinement search as a random heuristic search is that we can use the theoretical results from random heuristic search in order to analyse the behaviour and convergence of the search. In Definition 82, each state P_{i+1} of the search only depends on the previous state P_i. Hence, it can be shown that a random heuristic search can be described as a Markov chain. This property can be used to estimate the probability that a particular population is generated in the next iteration. Analyses of the convergence of different forms of random heuristic search (i.e. simulated annealing, genetic algorithms, etc.), have been discussed in [Vos99]. These analyses can also be applied to corresponding stochastic refinement searches. In the following we give an example for a stochastic search in the form of simulated annealing.

Example 42 *SFoil [PKK93] is a top-down stochastic ILP system which combines Foil with a stochastic search in the form of simulated annealing. The stochastic search is used to choose the next literal to be added to the body of the clause. Hence, the refinement in SFoil can be viewed as downward unary stochastic refinement. The behaviour of the search can be analysed using the framework of random heuristic search. The following analysis is adapted from [Vos99]. The sample size for simulated annealing is* $s = 1$ *and given a population (i.e. probability vector* p*) and an objective function* f*, the next population (i.e. probability vector* q*) is obtained by the following stochastic procedure: 1) sample* q *from a neighbourhood* $N(p)$ *of* p *2) if* $f(q) < f(p)$ *then the next generation is* q*, otherwise the next generation is* q *with probability* $e^{(f(p)-f(q))/T_t}$ *where* T_t *is the temperature at generation* t*. Then the heuristic function* h *which determines the stochastic transition between two distinct states* i, j *is defined as follows:* $h(t,j)_i = \frac{[i \in N(j)]}{|N(j)|}([f(i) < f(j)] + [f(i) \geq f(j)]e^{(f(j)-f(i))/T_t})$ *where* $[expr]$ *returns 1 if* $expr$ *is true and 0 otherwise.*

6.3 Genetic search for learning first-order clauses

In this section we discuss Genetic Algorithms (GAs) as search methods for learning first-order clauses. GAs are among the most successful forms of stochastic methods which have been applied to many combinatorial and computationally hard problems [Ala08]. It has been shown [VL91] that GAs are instances of the framework of

```
SGA - Simple Genetic Algorithm
 Input: Fitness function $f(x)$, chromosome length $l$,
          crossover probability $p_c$, mutation probability $p_m$
 Output: Best individual
 1  Start with a (random) population of binary strings of lento $l$
 2  Calculate the fitness $f(x)$ of each string $x$ in the population
 3  Choose (with replacement) two parents from the current population with
          probability proportional to each string's relative fitness in the population
 4  Cross over the two parents (at a single randomly chosen point) with
          probability $p_c$ to form two offspring
 5  Mutate each bit in the offspring with probability $p_m$ and
          Place them in the new population
 6  Go to step 2 until a new population is complete
 7  Go to step 1
```

Figure 6.6: A variant of Simple Genetic Algorithms (SGA) adapted from [Mit98].

random heuristic search (Definition 82) and mathematical models of GAs based on this framework have been discussed in [Vos95] and [Vos99]. In this section we first review some properties of GAs and then discuss several learning systems which use genetic and evolutionary approaches for learning first-order clauses.

GAs are search methods which can solve hard combinatorial problems using an approach based on the mechanics of natural selection and genetics [Hol75, Gol89]. The main idea of GAs can be summarised as follows. Given a problem to solve, GAs generate a number of random (or informed) initial solutions, evaluate each solution at solving the problem in hand, then combine and mutate the fittest solutions to produce the next generation of solutions. The process continues until an acceptable solution is found, or no further improvement is possible. Figure 6.6 shows a variant of Simple Genetic Algorithms (SGA) adapted from [Mit98].

Even though each individual genetic operator (i.e. selection, recombination and mutation) may not be particularly interesting as a search operator, it is the combination of these operators which promote a useful search. For example, it has been suggested that the combination of selection and mutation contributes to continual improvement and the combination of selection and recombination contributes to cross-fertilising innovation [Gol02].

The representation and operators in GAs are closely related and must be designed as a whole [Bäc95]. Conventional GAs usually require formulating problem solutions in such a way that they can be processed by genetic operators. It has been suggested that the binary representation is a suitable coding for representing problem solutions in GAs [Hol75, Gol89]. This is known as *the principle of minimal alphabet* and even though this principle has been challenged (e.g. [Ant89]), a major part of the existing literature on GA, in particular the theoretical analyses assume a binary representation. The convergence of GAs is usually explained using Schema Theorem [Hol75]. A schema is a similarity template describing a subset of strings. Schemata are processed by a genetic algorithm and according to Schema Theorem, short, low-order and above average schemata receive at least exponentially increasing number of trials in successive generations. These highly fit, short, low order schemata (also known as building-blocks) become the partial solutions to a problem within the search. It has been also suggested [Hol75] that in each generation we perform computation proportional to n, the size of population, but we get useful processing on the order of $O(n^3)$ schemata. This is known as *Implicit Parallelism*. There are also extensions of Schema Theorem which do not assume a binary representation (e.g. [Vos91] and [Rad91]).

One important feature of GAs which makes them different from other search methods is that they tend to rely on recombination (also known as crossover) as the principal search mechanism. This is especially useful in many problems, including ILP, where the solutions feature recombinable building blocks (i.e. partial solutions). Note that GAs are instances of random heuristic search and if the genetic operators assume a generality order then GAs can also be viewed as stochastic refinement search as described in Section 6.2. In this case, crossover and mutation can be viewed as binary and unary stochastic refinement operators respectively as shown in Figure 6.7.

The recombination operator also gives more exploration power to GAs in order to work as a global strategy compared to most stochastic search methods which use local strategies that step only to neighboring states. Another important feature of GAs is that they are less dependent on biases and heuristics which are needed by most deterministic search methods. For example it has been shown [GS92] that a GA-based learning system can easily solve some classes of learning problems in which the correct solution cannot be located by similar non-GA systems which use information gain

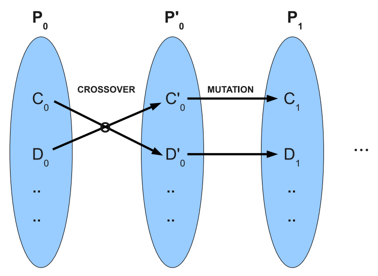


Figure 6.7: Simple Genetic Algorithm (SGA) as a stochastic refinement search with binary (crossover) and unary (mutation) stochastic refinement operators

heuristics [Qui90]. GAs are population-based methods and search from a population of points, not a single point and therefore they are less sensitive to local optima. This however contributes to a higher computational cost of GAs compared to a non-population-based method, especially when the problem does not involve a complex search. This can be compensated by the fact that GAs are highly parallel. In addition to the implicit parallelism described above, GAs are naturally suitable for parallel implementation [Ste93, CP00]. This makes it easier to benefit from the computational power of parallel and distributed hardware. GAs therefore have great potential to scale-up for search intensive problems. On the other hand, GAs are syntactically restricted and cannot represent a priori knowledge that already exists about the domain. By contrast, first-order concept learning methods, such as ILP, benefit from the expressive power inherited from logic and logic programming.

These complementary features of GAs and first-order learning motivated some authors to investigate combinations of GAs and first-order learning. Some of these systems (e.g. [GS92, GN96, Hek98, AGLS98]) follow conventional genetic algorithms and represent problem solutions by fixed length bit-strings. Other systems (e.g. [Var93, LW95, KGC99]) use a hierarchical representation and evolve a population of logic programs in a Genetic Programming (GP) [Koz91] manner.

Table 6.1 compares some of logic-based learning systems which use some form of genetic search (e.g. GA or GP). A summary of each system, in particular features which are relevant to the implementation and evaluation of algorithms in this thesis (e.g. representation, operators and fitness function) are discussed in Appendix B. As shown in

System	Representation	Type of EC	Use of BK
GA-SMART [GS92] REGAL [GN96] G-Net [AGLS98]	Binary Encoding	Distributed GA	Manual language template
DOGMA [Hek98]	Binary Encoding	Distributed GA	Manual template & speciation
SIAO1 [SAK95]	Direct	SGA-like	Hierarchy of concepts
GLPS [LW95] LOGENPRO [WL97]	Program Tree	GP	Seeding the population
STEPS [KGC99]	Program Tree	STGP	Seeding the population

Table 6.1: A comparison between some of logic-based learning systems which use a genetic search. A summary of each system is discussed in Appendix B.

this table, REGAL, DOGMA and G-NET follow the same basic idea as in GA-SMART and employ a language template for mapping first-order rules into bit strings. A language template is a fixed length CNF formula which can be (manually) constructed from the background knowledge. Mapping a formula into a bit-string is done by setting the corresponding bits to represent the occurrences of predicates in the formula. These systems need to adapt a non-standard first-order representation based on a template which is a conjunction of internally disjunctive predicates. The main problem of this approach is that the number of conjuncts grows combinatorially with the number of predicates. The template, therefore, can be very large in some circumstances. Difficulties related to providing the template by the user is also a major disadvantage of this approach. Even though the template can include parts of background knowledge, this needs to be done manually. DOGMA uses background knowledge to also divide chromosomes into species. Similarly, SIAO1 uses a simple form of background knowledge to establish a hierarchy between concepts. SIAO1 uses a direct mapping for representing first-order rules which means that no binary encoding is required. However, the genetic operators are limited because of the direct representation and this could lead to a low diversity population. GLPS, LOGENPRO and STEPS use hierarchical representations (program trees) rather than fixed length bit-strings and background knowledge is used for seeding the population. In general GP-based systems need to deal with a larger and more complex search space compared to the GA-based systems.

In summary these systems use a limited form of background knowledge to seed the population or classify the chromosomes and they cannot benefit from intentional background knowledge in the same way as in ILP systems. This is mainly because in these systems genetic search has been used as the main and the only learning mechanism.

In Section 6.4 we discuss a hybrid GA-ILP framework in which background knowledge can be used in the same way as in the ILP systems such as Progol and Aleph. This GA-ILP framework not only uses a standard ILP representation but we also show that the proposed encoding is isomorphic to the bounded subsumption lattice. This property can be used to design task-specific genetic operators.

6.4 Genetic and stochastic refinement search for bounded subsumption

As discussed in Section 6.3, most of the existing logic-based learning systems that use a genetic search for learning first-order clauses cannot benefit from intentional background knowledge in the same way as in an ILP system. In particular, GA-based systems (e.g. G-Net) require a language template which is (manually) extracted from background knowledge. In this thesis we describe a genetic and stochastic refinement approach in which a template (i.e. a bottom clause) is automatically constructed using ILP methods (e.g. Inverse Entailment). Moreover, unlike the GA-based systems discussed in Section 6.3, we use a standard logic-based representation. We also show that the encoding and operators can be interpreted in terms of well-known ILP concepts, e.g. subsumption. This hybrid GA-ILP framework is the basis of several implementations, i.e. GA-Progol, QG, QG/GA and ProGolem. The details of these implementations are described in Chapter 7. In this section we briefly review the main features of these systems which can be summarised as the following two stages:

Stage 1: Construction of $\bot_e$ using ILP In the first stage a bottom clause is constructed using an ILP method (e.g. Inverse Entailment). This bottom clause defines a bounded subsumption lattice. The lattice structure and refinement operators for the bounded subsumption were studied in Chapters 4 and 5. It was also shown (See Section 5.3) that, unlike for the general subsumption order, efficient *lgg*-like operators can be implemented for the subsumption relative to a bottom clause (e.g. $lgg_\bot$ and $armg_\bot$).

Stage 2: Stochastic search for bounded subsumption In the second stage a stochastic search algorithm is used to find the best hypotheses from clauses in the bounded subsumption lattice. This search algorithm could be either a standard random heuristic search (e.g. SGA-like search in GA-Progol) or a monotonic random heuristic search (Definition 83) which can be viewed as a stochastic refinement search as discussed in Section 6.2 (e.g. Golem and ProGolem). In this case, stochastic refinement operators can be used as discussed in Section 6.1.

Note that in the second stage mentioned above, both unary refinement operators (e.g. ρ_1 and ρ_2 described in Chapter 5) and binary refinement operators relative to bounded subsumption (e.g. $lgg_\perp$) can be used in the stochastic search. However, it has been shown [MM99] that the minimum depth of any clause within the binary refinement graph is logarithmically related to the depth in the refinement graph of corresponding unary operator whenever a certain lattice divisibility assumption is met. This result suggests that a stochastic refinement search which uses a binary refinement operator relative to $\perp$ (e.g. $lgg_\perp$) requires logarithmically fewer refinement steps to find a target clause compared to the case when a unary refinement operator is used. In other words, when using refinement relative to a bottom clause, a binary stochastic refinement could lead to logarithmic convergence relative to a unary stochastic refinement. The empirical results reported for ILP systems such as GA-Progol which uses some forms of binary stochastic refinement relative to a bottom clause are consistent with the logarithmic convergence described above. These empirical results are discussed in Chapter 8.

GA-Progol [TNM02] is a stochastic ILP system in which the A^*-like search of Progol is replaced by a genetic algorithm. GA-Progol can be described based on the two-staged framework described in this section. The stochastic refinement in GA-Progol includes both unary and binary stochastic refinement. GA-Progol also uses stochastic lgg and mgs operators relative to a bottom clause ($lgg_\perp$ and $mgs_\perp$). The details of GA-Progol are described and discussed in Chapter 7.

6.5 Related work and discussion

The refinement graph theory has been viewed as the main theoretical foundation of ILP [NCdW97]. Since the publication of this theory, there have been attempts to build ILP systems based on stochastic and randomised methods (e.g. [PKK93, Sri00, TNM02, RK03, ZSP06, PŽZ$^+$07, MTN07, DPZ08]). However, to date there is very little theory to support the developments of these systems. We believe the theoretical part on stochastic refinement presented in this chapter is a step in this direction.

In the following we discuss examples of ILP systems which use some form of stochastic refinement. Aleph [Sri07] implements several randomised search methods including randomised local searches such as GSAT, WalkSAT, simulated annealing and also a randomised rapid restart search [ZSP06]. Randomised local searches in Aleph can be characterised as stochastic refinement search. Note that in some cases the stochastic refinement operators in Aleph are bi-directional and therefore the corresponding stochastic searches are non-monotonic. Golem [MF90] is a bottom-up ILP system which employs determinate least general generalisation under relative subsumption (RLGG). Golem combines random sampling and a hill-climbing search to construct RLGGs with new randomly sampled positive examples at each iteration. The refinement in Golem can be viewed as an upward stochastic refinement. As shown in Example 40, a proper sample size can be estimated so that one could guarantee that with a high probability a consistent *RLGG* can be generated at each iteration. The *RLGGs* are constructed with new sampled examples at each iteration until the search converges to a set of consistent clauses each covering a partition of positive examples. SFoil [PKK93] is a top-down stochastic ILP system which combines Foil with a stochastic search in the form of simulated annealing. The stochastic search is used to choose the next literal to be added to the body of the clause. Hence, the refinement in SFoil can be viewed as downward unary stochastic refinement. Similarly, the simulated annealing framework for ILP described in [SPR04b] can be characterised using stochastic refinement search. The approach based on estimation distribution algorithms described in [PZ12] is closely related to the concepts of stochastic refinement as well as random heuristic search (i.e. using distributions rather than explicit individuals) described in this chapter. This work is also related to stochastic refinement search for bounded subsumption as it uses (reduced) bottom clauses.

GA-Progol [TNM02], Quick Generalisation (QG) algorithm and QG/GA [MTN07] and ProGolem [MSTN10a] are also relevant to stochastic refinement search described in this chapter. These algorithms are discussed in the next chapter.

6.6 Summary

In this chapter we discussed how the refinement theory and relevant concepts such as refinement operators can be adapted for a stochastic ILP search. To address this question we introduced the concept of stochastic refinement operators and adapted a framework, called stochastic refinement search. Stochastic refinement is introduced as a probability distribution over a set of clauses which can be viewed as a prior in a stochastic ILP search. We gave an analysis of stochastic refinement operators within the framework of stochastic refinement search. We studied the properties of a stochastic refinement search as two well known Markovian approaches: 1) a Gibbs sampling algorithm and 2) a random heuristic search. As a Gibbs sampling algorithm, a stochastic refinement search iteratively generates random samples from the hypothesis space according to a posterior distribution. In an ILP setting, we need to make sure that at least one consistent and compressive clause can be generated at each iteration. We have shown that a minimum sample size can be set so that in each iteration a consistent clause is generated with a high probability. We defined a special case of random heuristic search [Vos99] called monotonic random heuristic search. A stochastic refinement search can be viewed as a monotonic random heuristic search. The advantage of studying stochastic refinement search as a random heuristic search is that we can use the theoretical results from random heuristic search in order to analyse the behaviour and convergence of the search.

In this chapter we also discussed genetic search for learning first-order clauses and describe a GA-ILP framework for genetic and stochastic refinement search for bounded subsumption. This framework is the basis of the algorithms which are described in the next section.

Chapter 7

Algorithms and implementations

In this chapter we describe stochastic algorithms for searching the hypothesis space bounded by a bottom clause. These algorithms are based on the theoretical results and properties of bounded subsumption from previous chapters. In particular we discuss a genetic algorithm approach which uses the encodings and operators for the bounded subsumption (from Chapter 5) and can also be characterised as a stochastic refinement search (Chapter 6). This genetic algorithm is implemented in GA-Progol, an extension of the ILP system Progol in which the standard refinement and the A^*-like algorithm for searching the bounded subsumption lattice is replaced by a genetic search. In addition to the genetic algorithm, in this chapter we also describe other stochastic refinement searches including Quick Generalisation (QG) algorithm and QG/GA search which are implemented in GA-Progol and algorithms based on Asymmetric Relative Minimal Generalisation (ARMG) which are implemented in ProGolem.

This chapter is organised as follows. GA-Progol is discussed in Section 7.1. This includes a SGA-like algorithm, representation, encoding, genetic operators and stochastic refinement in GA-Progol. QG and QG/GA algorithms are described in Section 7.2. ProGolem and algorithms based on ARMG are described in 7.3. Related work is discussed in Section 7.4 and Section 7.5 summarises the chapter.

7.1 GA-Progol

GA-Progol [1] is an extension of the ILP system Progol in which the standard refinement and the A^*-like algorithm for searching the bounded subsumption lattice is replaced by stochastic refinement in the form of a genetic algorithm. Progol's standard refinement setting was described in Chapter 3 and can be summarised as follows.

Definition 84 (Progol refinement setting) *Let $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\perp \rangle$ be Progol's ILP setting where $\mathcal{B}$ is the background knowledge, E is the set of examples, $\mathcal{L}$ be Progol's definite language and $\succeq_\perp$ subsumption relative to a bottom clause as defined in Definition 55. Let $E = \langle E^+, E^- \rangle$ consist of a set of positive and negative examples (ground unit clauses) respectively. The top clause, denoted by $\triangle$, is the maximal$_{\succeq_\perp,\mathcal{L}}$ element in $\mathcal{L}$. The bottom clause, denoted by $\perp_e$, is the least$_{\succeq_\perp,\mathcal{L}}$ element such that $\mathcal{B}, \perp_e \models e$. Refinement of clause C, denoted by $\rho(C)$, is the set of maximal$_{\succeq_\perp,\mathcal{L}}$ clauses D such that $C \succ_\perp D \succeq_\perp \perp_e$ as defined in Definition 45.*

GA-Progol's set-covering algorithm is shown in Figure 7.1. This is similar to Progol's set-covering algorithm, however, in GA-Progol a stochastic refinement setting can be selected instead of Progol's standard refinement setting. The set-covering algorithm repeatedly constructs a bottom clause from the next positive example e in E^+ and then searches for the best clause C in the subsumption lattice bounded by $\perp_e$. C is then added to hypothesis H and positive examples which are covered by $B \wedge H$ are removed from E^+ for the next iterations. The algorithm terminates in at most $|E^+|$ iterations.

The algorithm for constructing $\perp_e$ is given in Appendix A. The time-complexity of constructing $\perp_e$ is proportional to the cardinality of $\perp_e$.

Theorem 22 (Cardinality of $\perp_e$ [Mug95]) *Let h, i, B, M and $\perp_e$ be defined as in Definition 43 and let $|M|$ denote the cardinality of mode declaration M. Let the number of +type and -type occurrences in each modeh in M be bounded by constants j^- and j^+ respectively. Let the recall of each m in M be bounded by the constant r.*

[1] Available from http://ilp.doc.ic.ac.uk/GA-Progol/

```
GA-Progol's cover-set algorithm
 Input: Examples $E$, mode declarations $M$, background knowledge $B$,
        search_mode ($A^*$, $GA$, $QG$ or $QG/GA$)
 Output: Hypotheses $H$
 01 $H := \emptyset$
 02 $E^+ :=$ positive examples in $E$
 03 while $E^+ \neq \emptyset$
 04   $e :=$ first example in $E^+$
 05   Construct the bottom clause $\bot_e$ from $e$, $M$ and $B$ (See Appendix A)
 06   $C :=$ best_clause($\bot_e$, $E$, search_mode)
 07   if compression($C$) $> 0$
 08     $H := H \cup C$
 09     $E^+_H :=$ positive examples covered by $B \wedge H$
 10     $E^+ := E^+ - E^+_H$
 11   end if
 12 end while
 13 return $H$
```

Figure 7.1: GA-Progol's cover-set algorithm.

Then the cardinality of $\bot_e$ is bounded by $(r|M|j^+j^-)^{ij^+}$.

The proposition below follows from Theorem 22.

Proposition 23 *Let h, i, B, M and $\bot_e$ be defined as in Definition 43, $\mathcal{L}_i(M)$ be a depth-bounded mode language as defined in Definitions 42, i the maximum variable depth in $\mathcal{L}_i(M)$ and j be the maximum arity of any predicate in M. Then the length of $\bot_e$ is polynomially bounded in the number of mode declarations in M for fixed values of i and j.*

As shown in Figure 7.1, user defined input parameter *search_mode* determines which search strategy to be used, i.e. A^*, GA, QG or QG/GA. If A^* is selected then *best_clause* (step 6) simply calls Progol's standard A^*-like search as described in Appendix A. If other search methods are selected then *best_clause* uses the SGA-like or QG/GA algorithms to find the most compressive clause in the lattice defined by $\bot_e$. SGA-like algorithm is described in Section 7.1.1 and QG and QG/GA algorithms are described in Section 7.2.

7.1.1 SGA-like algorithm

GA-Progol adapts a variant of genetic algorithms known as Simple Genetic Algorithm (SGA) as described in [Gol89]. Figure 7.2 shows the *SGA*-like search implemented in GA-Progol. The *SGA*-like search uses a binary representation for encoding clauses in the bounded subsumption lattice. Depending on the *encoding_mode*, a binary string in the population encodes the variable bindings or the occurrences of literals in a clause with respect to a bottom clause. This binary encoding is based on the theoretical results from Chapters 4 and 5 which indicate that clauses in the bounded subsumption lattice can be represented by a set of variable partitions relative to $\perp$ (e.g. see Propositions 16 and 19). In particular θ and K in the encoding tuples described in Chapter 5 are implemented using binary strings. This binary representation is discussed in Section 7.1.2. As shown in Figure 7.2, in each run of the *SGA*-like search an initial population of binary strings are generated and evolved using genetic operations, i.e. selection, crossover and mutation. The fitness value of each individual is similar to the evaluation criteria used in the A^*-like search implemented in Progol, i.e. *compression* which can be defined as $compression = p-n-h$, where p is the number of observations correctly explained by the hypothesis, n is the number incorrectly explained and h is the length of the hypothesis. The initial population can be generated at random, i.e. by generating random binary strings as in standard genetic algorithms, or by sampling from consistent clauses using Quick Generalisation (QG) algorithm. QG algorithm is described in Section 7.2. In this section we assume that the initial population is generated at random. At each generation of the *SGA*-like search, new individuals are generated by selecting a pair of parental individuals using *select* function which stochastically select individuals, e.g. with a probability proportional to their fitness values. This pair of parental strings are re-combined using the crossover operation, and the resulting child strings are then mutated and added to the new population. At each generation all new individuals are decoded into corresponding clauses and then clauses are evaluated. The best individual with the highest fitness value is then compared with the best individual from previous generation and after reaching a pre-defined number of generations ($maxgen$) the clause corresponding to best fit individual is returned as output. The representation and encoding are discussed in Section 7.1.2. The details

SGA-like search

Input: Examples E, bottom clause $\bot_e$, population size *popsize*, maximum generation *maxgen*, *initpop_mode* (*rand* or *QG*)

Output: Clause C with highest fitness (i.e. compression)

```
01 $gen := 0$
02 $currentpop$ := init_pop($\bot_e$, $E$, $initpop\_mode$) % generate and evaluate initial pop.
03 $current\_bestfit$ := max_fitness($currentpop$) % best individual in $currentpop$
04 while $gen < maxgen$
05   $j := 0$
06   while $j < popsize - 1$
07     $mate_1$ := select($currentpop$) % stochastically select a pair of individuals
08     $mate_2$ := select($currentpop$)
09     crossover($mate_1$, $mate_2$, $child_1$, $child_2$) % crossover $mate_1$ and $mate_2$
10     mutation($child_1$) % mutate new individuals
11     mutation($child_2$)
12     $clause_1$ := decode($child_1$) % decode new individuals
13     $clause_2$ := decode($child_2$)
14     $fitness_1$ := evaluate($clause_1$, $E$) % evaluate new individuals
15     $fitness_2$ := evaluate($clause_2$, $E$)
16     $newpop[j].chrom := child_1$, $newpop[j+1].chrom := child_2$
17     $newpop[j].clause := clause_1$, $newpop[j+1].clause := clause_2$
18     $newpop[j].fitness := fitness_1$, $newpop[j+1].fitness := fitness_2$
19     $j := j + 2$ % increment population index
20   end while
21   $new\_bestfit$ := max_fitness($newpop$) % best individual in $newpop$
22   if ($new\_bestfit.fitness > current\_bestfit.fitness$) then
23     $current\_bestfit := new\_bestfit$
24   $currentpop := newpop$ % advance the generation
25   $gen := gen + 1$
26 end while
27 $C := current\_bestfit.clause$
28 return $C$
```

Figure 7.2: *SGA*-like search in GA-Progol.

a)

eastbound(A):-	has_car(A,B)	has_car(A,C)	closed(C)	open(B)	short(C)	. . .

b)

	0	1	1	0	1	. . .
eastbound(A):-		has_car(A,C)	closed(C)		short(C)	. . .

Figure 7.3: Binary encoding of the occurrences of literals in a clause with respect to a bottom clause. a) a bottom clause b) binary encoding of clause $eastbound(A) \leftarrow has\_car(A, C), closed(C), short(C)$.

of genetic operations, i.e. selection, crossover and mutation are discussed in Section 7.1.3.

7.1.2 Representation and encoding

In this section we introduce a novel binary representation for clauses in the bounded subsumption lattice. As shown in Chapter 5 (e.g. Definition 68), each clause $\overrightarrow{C}$ in the bounded subsumption lattice can be represented by a tuple $\langle K, \theta\rangle$, where K is a set of indexes from the bottom clause and θ is a variable subsumption which maps variables between $\overrightarrow{C}$ and $\overrightarrow{\bot_v}$. In this section we introduce a binary encoding for a tuple $\langle K, \theta\rangle$. The binary encoding of $\langle K, \theta\rangle$, and therefore $\overrightarrow{C}$, can be represented by a tuple $\langle V, M\rangle$, where $V = v(K)$ is an occurrence vector and $M = m(\theta)$ is a binding matrix as defined in the following Definitions.

Definition 85 (Occurrence vector) *Let $\overrightarrow{\bot_v}$ be as defined in Definition 53, n be the number of literals in $\overrightarrow{\bot_v}$, $\mathcal{K}$ be the power set of $\{1, \ldots, n\}$ and $K \in \mathcal{K}$ as defined in Definition 68. The occurrence vector of K, denoted by $v(K)$, is an n-bit binary vector V in which V_i is 1 if $i \in K$ and V_i is 0 otherwise. $\mathcal{V}_n$ is the set of all n-bit binary vectors. We also represent V_i by $V[i]$.*

Example 43 *Consider the problem of learning Michalski's east-bound trains [Mic80]. Given the appropriate mode declaration, a bottom clause can be generated similar to the following clause:*

$$\begin{aligned} eastbound(A) \;\leftarrow\; & has\_car(A, B), has\_car(A, C), closed(C), open(B), short(C), \\ & wheels(B, 2), infront(B, C), wheels(C, 2), load(B, rectangle, 3), \\ & load(C, triangle, 1), shape(C, rectangle), \ldots \end{aligned}$$

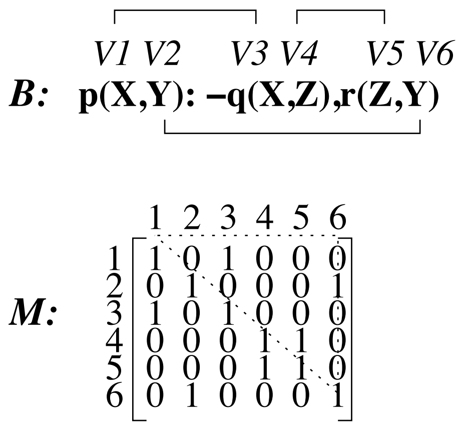


Figure 7.4: Binding matrix for clause *p(X,Y):-q(X,Z),r(Z,Y)*.

As shown in Figure 7.3, the occurrences of literals in a clause is encoded as a bit-string such that a '1' bit shows that the corresponding atom from the bottom clause occurs in the clause.

Definition 86 (Binding matrix) *Let $\overrightarrow{\bot_v}$ be as defined in Definition 53 and Θ be defined as in Definition 68. Let $\overrightarrow{\bot_v}$ have m variable occurrences representing variables $\langle v_1, v_2, \ldots, v_m\rangle$ and θ be a variable substitution in Θ. The binding matrix of θ, denoted by $m(\theta)$, is an $m \times m$ binary matrix M in which M_{ij} is 1 if $v_j/v_i \in \theta$ and M_{ij} is 0 otherwise. An $m \times m$ binary matrix M is in the set of normalised binding matrices $\mathcal{M}_m$ if M is symmetric and for each $1 \le i \le m$, $1 \le j \le m$ and $1 \le k \le m$, $M_{ij} = 1$ if $M_{ik} = 1$ and $M_{kj} = 1$. We also represent M_{ij} by $M[i,j]$.*

Example 44 *Let $\overrightarrow{\bot_v}$ be $p(V_1, V_2) \leftarrow q(V_3, V_4), r(V_5, V_6)$ and $\overrightarrow{C}$ be $p(X,Y) \leftarrow q(X,Z)$, $r(Z,Y)$. Then $\overrightarrow{C}$ can be represented by $\langle K, \theta\rangle = \langle\{1,2,3\}, \{V_3/V_1, V_5/V_4, V_6/V_2\}\rangle$. Variable bindings in $\overrightarrow{C}$ can be represented by a binary matrix as shown in Figure 7.4.*

The following mapping function d maps a tuple $\langle V, M\rangle$ into an ordered clause in $\overrightarrow{\mathcal{L}}_\bot$.

Definition 87 (Mapping function d) *Let $\mathcal{K}$ and Θ and mapping function c be defined as in Definition 68 and $K \in \mathcal{K}$ and $\theta \in \Theta$. Let $\mathcal{V}_n$ and v be defined as in Definition 85 and $\mathcal{M}_m$ and m be defined as in Definitions 86 and $V \in \mathcal{V}_n$ and $M \in \mathcal{M}_m$ such that $V = v(K)$ and $M = m(\theta)$. The mapping function $d : \mathcal{V}_n \times \mathcal{M}_m \to \overrightarrow{\mathcal{L}}_\bot$ is defined* *as follows:* $d(\langle V, M\rangle) = c(\langle K, \theta\rangle)$

In the following we first define the order relation for the binary encoding tuples $\langle V, M\rangle$ and show the relationship with the tuples $\langle K, \theta\rangle$. Then, we show that the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp\rangle$ and the lattice of the encoding tuples $\langle K, \theta\rangle$ described in Chapter 5 can also be extended to the binary encoding tuples $\langle V, M\rangle$.

Definition 88 *Let $\mathcal{V}_n$ and $\mathcal{M}_m$ be defined as in Definitions 85 and 86 and $V_1, V_2 \in \mathcal{V}_n$ and $M_1, M_2 \in \mathcal{M}_m$. $\langle V_1, M_1\rangle \subseteq \langle V_2, M_2\rangle$ if and only if $V_1 \subseteq V_2$ and $M_1 \subseteq M_2$. $V_1 \subseteq V_2$ if and only if for each $1 \leq i \leq n$, $V_1[i]$ is 1 if $V_2[i]$ is 1. $M_1 \subseteq M_2$ if and only if for each $1 \leq i \leq n$ and $1 \leq j \leq n$, $M_1[i, j]$ is 1 if $M_2[i, j]$ is 1.*

The proposition below follows from Definitions 85 and 86.

Proposition 24 *Let $\mathcal{K}$ and Θ be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. Let $\mathcal{V}_n$ and v be defined as in Definition 85 and $\mathcal{M}_m$ and m be defined as in Definitions 86 and $V_1, V_2 \in \mathcal{V}_n$ and $M_1, M_2 \in \mathcal{M}_m$ such that $V_1 = v(K_1)$, $V_2 = v(K_2)$, $M_1 = m(\theta_1)$ and $M_2 = m(\theta_2)$. $\langle K_1, \theta_1\rangle \subseteq \langle K_2, \theta_2\rangle$ if and only if $\langle V_1, M_1\rangle \subseteq \langle V_2, M_2\rangle$.*

The following theorem shows the ordering relationship between binary encoding and the bounded subsumption of clauses.

Theorem 23 *Let $\mathcal{V}_n$, $\mathcal{M}_m$ and mapping function d be defined as in Definition 87 and $V_1, V_2 \in \mathcal{V}_n$ and $M_1, M_2 \in \mathcal{M}_m$. $d(\langle V_1, M_1\rangle) \succeq_\perp d(\langle V_2, M_2\rangle)$ if and only if $\langle V_1, M_1\rangle \subseteq \langle V_2, M_2\rangle$.*

Proof. The proof follows directly from Theorem 17 and Proposition 24. □

According to Theorem 23, the ordering between clauses in the bounded subsumption lattice can be realised by the subset ordering between the binary encoding of the clauses. This property can be exploited by a genetic algorithm for searching the bounded subsumption lattice. In particular, this encoding defines a structured search space similar to refinement graph search used in ILP. This can also be used by genetic operators which we will discuss in the next section. A binding matrix is a symmetric

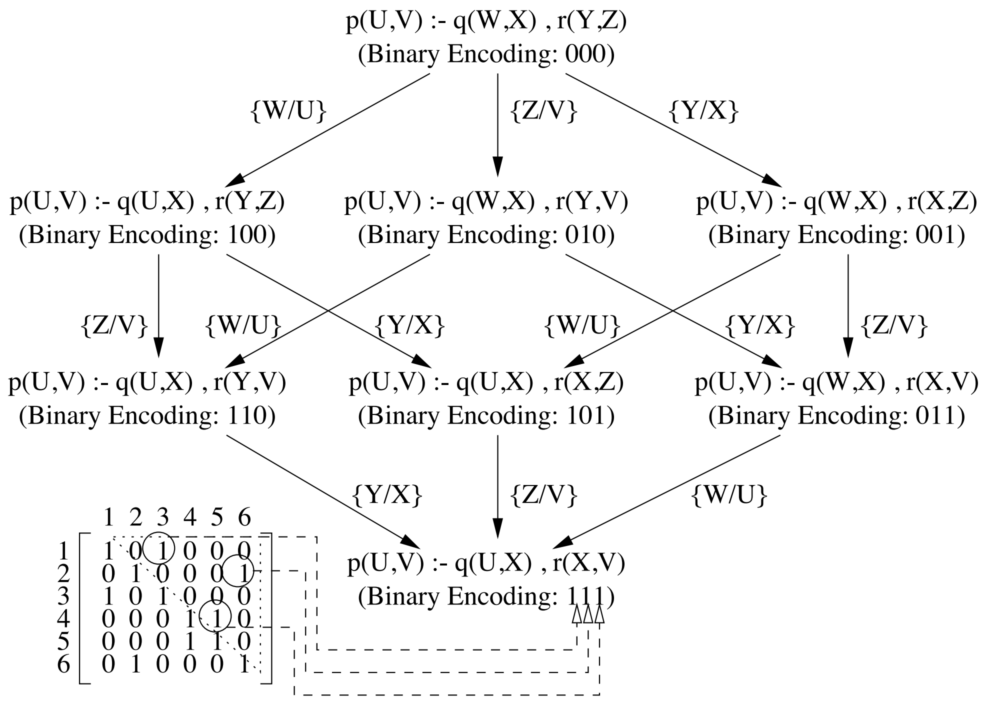


Figure 7.5: Encoding of variable bindings for clauses in a subsumption lattice bounded below by clause *p(U,V):-q(U,X),r(X,V)*.

matrix and we only need to maintain entries in top (or down) triangle of the matrix. Furthermore, we are interested in a subsumption lattice bounded below by a particular clause. Hence, the variable bindings for clauses in the bounded subsumption lattice can be encoded by a bit-string in which each bit corresponds to a 1 entry of the binding matrix for the bottom clause.

Example 45 *Figure 7.5 shows encoding of variable bindings for clauses in a subsumption lattice bounded below by clause* p(U,V):-q(U,X),r(X,V). *Each clause in this lattice can be encoded by 3 bits.*

Figure 7.6 shows the algorithms for generating and evaluating the initial population in the SGA-like search and also the algorithm for decoding a binary string into a clause. In *init_pop*, a population of binary strings is generated. Depending on *encoding_mode*, the binary strings represent either the occurrence vectors or binding matrices, *chromsize* is therefore either the length of the bottom clause or the number of variable positions

init_pop - generate and evaluate initial population

```
Input: Examples $E$, bottom clause $\perp_e$, $initpop\_mode$ ($rand$ or $QG$),
       $encoding\_mode$ ($atoms\_only$ or $var\_splitting$), probability of active bits $p_a$
Output: evaluated initial population $pop$
01 if $encoding\_mode = atoms\_only$ then
02   $chromsize$ := length($\perp_e$)
03 else
04   $chromsize$ := number of variable positions in $\perp_e$ sharing the same variable
05 let $bin\_str$ be a binary string with length $chromsize$
06 $pop := \{\}$
07 while $j < popsize$ do
08   set all bits of $bin\_str$ to 0
09   if $initpop\_mode = rand$ or $encoding\_mode = var\_splitting$ then
10     for $j = 1$ to $chromsize$
11       if flip($p\_active$) then % flip a biased coin
12         $bin\_str[j]$:= 1
13     $clause$ := decode($bin\_str$, $\perp_e$)
14     $fitness$ := evaluate($clause$, $E$)
15   else
16     $qg\_best$ := qg($\perp_e$, $E$) % see Figure 7.9
17     $bin\_str$ := $qg\_best.chrom$ ; $clause$ := $qg\_best.clause$ ; $fitness$ := $qg\_best.fitness$
18   end if
19   $pop[j].chrom$ := $bin\_str$ ; $pop[j].fitness$ := $fitness$ ; $pop[j].clause$ := $clause$
20   $j := j + 1$
21 end while
22 return $pop$
```

decode - decodes binary string into clause

```
Input: binary string $bin\_str$, bottom clause $\perp_e$,
       $encoding\_mode$ ($atoms\_only$ or $var\_splitting$), $ga\_lmode$
Output: clause $C$ decoded from $bin\_str$
01 if $encoding\_mode = atoms\_only$ then
02   let $C$ be an empty clause
03   add the head of $\perp_e$ to $C$
04   for $k = 1$ to $chromsize$
05     if $bin\_str[k] = 1$ then
06       add atom $k$ from $\perp_e$ to $C$
07 else
08   let $C$ be $\perp_e$ with all variable positions populated with new distinct variables ($\perp_v$)
09   for $k = 1$ to $chromsize$
10     let $i$ and $j$ be variable positions in the binding matrix of $\perp_e$ corresponding to $k$
11     if $bin\_str[k] = 1$ then
12       let variable positions $i$ and $j$ in $C$ have the same variable
13 filter_unconnected_atoms($C$, $ga\_lmode$)
14 return $C$
```

Figure 7.6: Algorithms for *init_pop* and *decode*

in the bottom clause sharing the same variable (as shown in Figure 7.5). Depending on *initpop_mode* the binary strings in the initial population are generated randomly or by the *QG* algorithm (see Section 7.2). Note that the *QG* algorithm can be only used for occurrence vectors (i.e. in *atoms_only* mode). For randomly generated binary strings, the probability that a bit is set to 1 is controlled by the user-defined parameter *p_active*. Each randomly generated binary string is first decoded into a clause. This clause is then evaluated and the fitness value for each new individual in the population is recorded. This process is repeated until a population with *popsize* individuals is generated. Binary strings can be decoded into clause using *decode*. In *atoms_only* encoding mode, a clause is constructed by adding literals of $\bot_e$ which correspond to '1' bits of the occurrence vector. For decoding binding matrices, '1' bits correspond to variable positions of $\bot_v$ which have the same variables. In both encoding modes, literals which are not 'head-connected' are filtered out from the decoded clause. Head-connectness is the property of clauses in $\mathcal{L}(M)$ (see Definition 41) and can be defined as follows:

Definition 89 (Head-connectness) *A definite clause $h \leftarrow b_1, .., b_n$ is said to be head-connected if and only if every input variable in any body atom b_i is either an input variable in h or an output variable in some body atom b_j, where $1 \leq j < i$.*

This definition can be implemented by maintaining a set of all valid input variables (i.e. input variables in the head and output variables in the previous atoms) and checking if all input variables of the current atom are in this set. This approach can be used for filtering invalid atoms (i.e. atoms with at least one invalid input variable) by scanning the clause from left to right. For clauses in $\overrightarrow{\mathcal{L}}_{\bot}$, which all follow the same ordering of literals as in $\bot$, the above mentioned scanning needs to be done only once, i.e. one pass through the clause. However, for clauses in $\overrightarrow{\mathcal{L}}^{i}_{\bot}$ (see Definition 74) where literals do not need to follow the same ordering of literals as in $\bot$, the above mentioned procedure should be repeated as a closure until there is no addition to the set of valid input variables. In the worst case this should be repeated n times where n is the length of the clause. This closure is necessary to consider input variables which could become valid in the next atoms. Both language modes are implemented in *filter_unconnected_atoms* and depending on user-defined parameter *ga_lmode*, either

$\overrightarrow{\mathcal{L}}_\perp$ (*ga_lmode*=1) or $\overrightarrow{\mathcal{L}}^i_\perp$ (*ga_lmode*=2) are selected.

7.1.3 Genetic operators and stochastic refinement

Genetic operators introduce new individuals into the population by randomly changing or combining the genotype of best-fit individuals during the evolutionary process. In conventional genetic algorithms these operators are domain-independent and usually without any assumption about the problem on hand. However, more efficient genetic operators can be designed by using simple facts about the domain. For example it has been shown that introducing generalisation and specialisation crossover operators, which are used together with standard crossover and mutation operators, can be useful when using GAs in concept learning problems [GS92, Jan93]. In this section we show that the binary representation, described in the previous section, has great potential for designing task-specific genetic operators. For example, we show that lgg (least general generalisation) and mgs (most general specialisation) for the bounded subsumption lattice, i.e. $lgg_\perp$ and $mgs_\perp$, can be implemented by simple bitwise operations on the binary encoding of clauses. In the following we first define bitwise operations $\wedge$ and $\vee$ for the binary encoding tuples $\langle V, M\rangle$.

Definition 90 *Let $\mathcal{V}_n$ and $\mathcal{M}_m$ be defined as in Definitions 86 and 85 and $V, V_1, V_2 \in \mathcal{V}_n$ and $M, M_1, M_2 \in \mathcal{M}_m$. $\langle V, M\rangle = \langle V_1, M_1\rangle \wedge \langle V_2, M_2\rangle$ if and only if $V = V_1 \wedge V_2$ and $M = M_1 \wedge M_2$. Similarly, $\langle V, M\rangle = \langle V_1, M_1\rangle \vee \langle V_2, M_2\rangle$ if and only if $V = V_1 \vee V_2$ and $M = M_1 \vee M_2$. $V = V_1 \wedge V_2$ if and only if for each $a_i \in V$, $b_i \in V_1$ and $c_i \in V_2$, $a_i = 1$ if $b_i = 1$ and $c_i = 1$ and $a_i = 0$ otherwise. $M = (M_1 \wedge M_2)$ if and only if for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ and $c_{ij} = 1$ and $a_{ij} = 0$ otherwise. Similarly, $V = V_1 \vee V_2$ if and only if for each $a_i \in V$, $b_i \in V_1$ and $c_i \in V_2$, $a_i = 1$ if $b_i = 1$ or $c_i = 1$ and $a_i = 0$ otherwise. $M = (M_1 \vee M_2)$ if and only if for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ or $c_{ij} = 1$ and $a_{ij} = 0$ otherwise.*

The proposition below follows from Definitions 85 and 86.

Proposition 25 *Let $\mathcal{K}$ and Θ and mapping function c be defined as in Definition 68 and $K_1, K_2 \in \mathcal{K}$ and $\theta_1, \theta_2 \in \Theta$. Let $\mathcal{V}_n$ and v be defined as in Definition 85, $\mathcal{M}_m$ and m be defined as in Definitions 86, mapping function d be defined as in Definition 87* *and $V_1, V_2 \in \mathcal{V}_n$ and $M_1, M_2 \in \mathcal{M}_m$ such that $V_1 = v(K_1)$, $V_2 = v(K_2)$, $M_1 = m(\theta_1)$ and $M_2 = m(\theta_2)$. Then, the following equalities hold:*

1. $c(\langle K_1, \theta_1\rangle \cap \langle K_2, \theta_2\rangle) = d(\langle V_1, M_1\rangle \wedge \langle V_2, M_2\rangle)$
2. $c(\langle K_1, \theta_1\rangle \cup \langle K_2, \theta_2\rangle) = d(\langle V_1, M_1\rangle \vee \langle V_2, M_2\rangle)$

The following theorem shows how $lgg_\perp$ and $mgs_\perp$ for clauses in the bounded subsumption lattice, can be implemented by bitwise $\wedge$ and $\vee$ operations on the binary encoding of clauses.

Theorem 24 *Let $\mathcal{V}_n$, $\mathcal{M}_m$ and mapping function d be defined as in Definition 87 and $V_1, V_2 \in \mathcal{V}_n$ and $M_1, M_2 \in \mathcal{M}_m$. Let $\wedge$ and $\vee$ operations be defined as in Definition 90. Then, the following equalities hold:*

1. $lgg_\perp(d(\langle V_1, M_1\rangle, d(\langle V_2, M_2\rangle) = d(\langle V_1, M_1\rangle \wedge \langle V_2, M_2\rangle)$

2. $mgs_\perp(d(\langle V_1, M_1\rangle, d(\langle V_2, M_2\rangle) = d(\langle V_1, M_1\rangle \vee \langle V_2, M_2\rangle)$

Proof. The proof follows directly from Proposition 20 and Proposition 25. □

Example 46 *In Figure 7.5, $lgg_\perp$ and $mgi_\perp$ of any pair of clauses in the lattice can be obtained by AND-ing ($\wedge$) and OR-ing ($\vee$) of their binary strings.*

According to Theorem 24, $lgg_\perp$ and $mgs_\perp$ for clauses in the bounded subsumption lattice can be done by simple bitwise operations on the binary encoding of clauses. This property can be used for implementing efficient task-specific genetic operators such as generalisation and specialisation crossover operators. Generalisation and specialisation are known as the main operations in concept learning methods [Win70, Mit82, Mit97]. In particular, common generalisation operators (e.g. lgg) are essential in logic-based machine learning. However, these operations could be inefficient in the general subsumption order. For example, with Plotkin's Relative Least General Generalisation (RLGG), clause length grows exponentially in the number of examples [Plo71]. Hence, an ILP system like Golem [MF90] which uses RLGG is constrained to ij-determinacy to guarantee polynomial-time construction. Golem [MF90] was successfully applied to several real-world applications (e.g. [BMV91, Fen92, MKS92]), however, the determinacy restrictions make it inapplicable in many key application areas, including the learning of chemical properties from atom and bond descriptions [Mug94]. On the other hand, with the bounded subsumption operators like $lgg_\perp$ and $mgs_\perp$ the clause length is bounded by the length of the initial bottom clause. Hence, these operators do not need the determinacy restrictions and as shown in Theorem 24, they can be implemented efficiently.

In a standard ILP setting, upward and downward refinement operators are used for generalisation and specialisation of clauses [NCdW97]. However, task-specific genetic operators can be defined as *stochastic refinement operators* as discussed in Chapter 6.

As mentioned in Section 7.2, at each generation of the SGA-like search, new individuals are generated by applying genetic operators on a pair of parental individuals which are stochastically selected from previous generation. Different selection mechanisms have been used in GAs, but two most popular selection mechanisms are i) fitness proportionate selection (also known as roulette-wheel selection) and ii) tournament selection. These two selection mechanisms are implemented in GA-Progol and the algorithms are shown in Figure 7.7. In fitness proportionate selection, normalised fitness values are computed for each individual, i.e. f/F where f is the fitness value of the individual and F is the sum of all fitness values for the population. Normalised fitness values are accumulated in variable *sum*. A number r between 0 and 1 is randomly selected and the first individual whose normalised fitness value makes *sum* greater than r is selected. Tournament selection works by running tournaments among individuals of a randomly chosen subset from the population as shown in Figure 7.7. The tournament size is a user-defined parameter which controls selection pressure, e.g. by increasing the tournament size individuals with low fitness value have a smaller chance to be selected. A pair of individuals returned by the selection mechanism are passed to the crossover and mutation operators.

Figure 7.8 shows algorithms for standard 1-point crossover and mutation operators. In standard 1-point crossover, the crossover operation is applied with probability p_c, otherwise the parental chromosomes strings are copied into child strings without any changes. If the crossover operation is applied, a crossover point *jcross* is randomly

select_rw - roulette-wheel selection

```
Input: population pop
Output: a stochastically selected individual from pop
01 let $r$ be a random number in $[0, 1)$
02 let $F$ be the sum of $pop[j].fitness$ for all $j$
03 $sum := 0$
04 for $i = 1$ to $popsize$
05   $f := pop[i].fitness$
06   $p_i := f/F$
07   $sum := sum + p_i$
08   if $r < sum$ then
09     return $pop[i]$
10   end if
11 end for
```

select_t - tournament selection

```
Input: population pop, tournament size tourn_size
Output: a stochastically selected individual from pop
01 let $tourn\_list$ be a randomly selected subset of $pop$
02 $winner := tourn\_list[1]$
03 for $i = 1$ to $tourn\_size$-1
05   $pick := tourn\_list[i + 1]$
06   if $pick.fitness > winner.fitness$ then
07     $winner := pick$
08   end if
09 end for
10 return $winner$
```

Figure 7.7: Algorithms for roulette-wheel selection and tournament selection. These algorithms are used to stochastically select an individual from a population.

crossover_1p - 1-point crossover

Input: parental strings $mate_1$ and $mate_2$, crossover probability p_c

Output: child strings $child_1$, $child_2$

```
01 if flip($p_c$) then % do crossover with probability $p_c$
02   $jcross$ := rnd(2, $chromsize$)
03   for $i = 1$ to $jcross - 1$
04     $child_1[i] := mate_1[i]$
05     $child_2[i] := mate_2[i]$
06   for $i = jcross$ to $chromsize$
07     $child_1[i] := mate_2[i]$
08     $child_2[i] := mate_1[i]$
09 else
10   for $i = 1$ to $chromsize$
11     $child_1[i] := mate_1[i]$
12     $child_2[i] := mate_2[i]$
```

mutation - mutate a binary string

Input: a binary string bin_str, mutation probability p_m

Output: mutated binary string bin_str

```
01 for $i$= 1 to $chromsize$
02   if flip($p_m$) then % mutate a bit with probability $p_m$
03     $bin\_str[i]$ := $bin\_str[i]$ xor 1
```

Figure 7.8: Algorithms for standard 1-point crossover and mutation operators.

selected between 2 and *chromsize*. Given parental bit strings $mate_1$ and $mate_2$, bits between 1 to $jcross1$ from $mate_1$ and $mate_2$ are copied to $child_1$ and $child_2$ respectively, and from $jcross$ to $chromsize$ bits of $mate_1$ and $mate_2$ are copied to $child_2$ and $child_1$ respectively. Standard mutation operator works by visiting every bit of a given string and changing it with probability p_m, i.e. if the bit is 1, it is changed to 0 and vice versa.

Note that when *encoding_mode* is set to *var_splitting*, a random crossover or mutation operators could generate individuals which are not normalised with respect to Definition 86. However, the decode algorithm described in Figure 7.6 still generates a valid clause in which the bindings between connected variables are set according to the last relevant bits in the binary encoding. However, the binary representation for variable bindings is redundant, i.e. there is a many-to-one genotype-to-phenotype mapping. A redundant representation is not regarded as a serious problem in GAs and some authors (e.g. [Alt95] and [RG03]) suggest that under some conditions a redundant representation can even improve the genetic search. The binary representation used here is one possible implementation of the partition-based encoding described in Chapter 5. The redundancy can be avoided using a different encoding (e.g. Grouping Genetic Algorithms (GGA) [Fal98]) as discussed in Section 9.3.

7.2 Quick Generalisation (QG) and QG/GA algorithm

Quick Generalisation (QG) is a stochastic algorithm which constructs maximally general consistent clauses by randomly pruning bottom clauses. The algorithm can be made arbitrarily efficient by choice of sample size (down to 1). This algorithm can be viewed as a stochastic refinement search algorithm as described in Chapter 6. A sampling mechanism based on QG as well as a combination of QG with a GA are explored in this section. These algorithms are implemented as part of GA-Progol [2].

Before describing the details of the QG algorithm, we introduce the notion of consistency within the Progol refinement setting.

Definition 91 ($\mathcal{S}$-consistency) *Let $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\perp \rangle$ and $E = \langle E^+, E^- \rangle$ be as de-*

[2] Available from http://ilp.doc.ic.ac.uk/GA-Progol/

```
qg - Quick Generalisation (QG) algorithm
Input: bottom clause $\bot_e$, examples $E$
Output: a randomly generated clause which is reduced wrt $E$
01 let $R$ be a random head-connected permutation of $\bot_e$
02 $C$ := reduce($R$, $E$)
03 return $C$
```

```
reduce - clause reduction algorithm
Input: clause $C = h \leftarrow b_1, .., b_n$, examples $E$
Output: reduced clause $Res$ from $C$
01 $Res := C$
02 while there is an unseen cutoff atom $b_i$ in the body of $Res$
03   for $b_i$ find minimal support set $S_i = \{b'_1, .., b'_m\} \subseteq \{b_1, .., b_{i+1}\}$
                 such that $h \leftarrow S_i, b_i$ is head-connected
04   $Res$ is $h \leftarrow S_i, b_i, S_{i-1}$ where $S_{i-1}$ is $b_1, .., b_{i-1}$ with $S_i$ removed
05 end while
06 return $Res$
```

Figure 7.9: Quick Generalisation (QG) algorithm.

fined in Definition 84, $e \in E^+$ and $\triangle \succeq_\bot C \succeq_\bot \bot_e$, where $\triangle$ and $\bot_e$ are the top and bottom clauses, respectively, as defined in Definition 84. We say that C is $\mathcal{S}$-consistent iff $\mathcal{B}, E, C$ is satisfiable.

In the following we define a minimal support set as an irreducible set of body atoms which ensure that a clause is head-connected (see Definition 89).

Definition 92 (Minimal support set) *Let $h \leftarrow B$ be a definite clause and B be a set of atoms. $S \subseteq B$ is a minimal support set for $b \subset B$ iff $h \leftarrow S, b$ is head-connected and there does not exist a set $S' \subset S$ for which $h \leftarrow S', b$ is head-connected.*

The notion of *fringe* is introduced as a set of maximally general clauses in the hypothesis space, from which QG samples.

Definition 93 (Fringe) *Let $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\bot \rangle$, $E = \langle E^+, E^- \rangle$ and ρ be as defined in Definition 84 and $e \in E^+$. Clause C is in Fringe(e,$\mathcal{S}$) iff it is head-connected and for every D it is the case that $C \in \rho(D)$ implies D is not $\mathcal{S}$-consistent.*

The QG algorithm (see Figure 7.9) works by finding successively smaller consistent subsets of a bottom clause. The notion of a cutoff atom is used within the algorithm to define the minimal consistent prefix of a given clause body.

Definition 94 (Profile and cutoff atom) *Let $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\perp \rangle$ and $E = \langle E^+, E^- \rangle$ be as defined in Definition 84 and $C = h \leftarrow b_1, \ldots, b_n$ be a definite clause in $\mathcal{L}$. $E_i \subseteq E^-$ is the ith negative profile of C, where $E_i = \{e : \exists \theta, e = h\theta, \mathcal{B} \models (b_1, \ldots, b_i)\theta\}$. b_i is the cutoff atom iff i is the least value such that $E_i = \emptyset$.*

As shown in Figure 7.9, the QG algorithm randomly permutes the given clause body and then applies to the result the *reduce* algorithm, a deterministic algorithm which uses negative examples to reduce a clause. This algorithm works by keeping only the literals which prevent negative examples from being proved [3]. The output of the QG is a randomly constructed fringe clause (see Definition 93).

Example 47 *Figure 7.10 illustrates the QG algorithm in the east-bound train problem. Given a random permutation of the bottom clause, in the first iteration of the reduce algorithm, the atom load(B,hexagon,1) is identified as the cutoff atom. This atom is placed immediately after the atom has_car(A,B) in order to ensure that variable B has been introduced before being used (i.e. the clause is head-connected). In the second iteration load(B,hexagon,1) is once more identified as the cutoff atom. Since this atom has already been seen, the algorithm terminates and returns the resulting clause in step 3.*

The correctness of the QG algorithm is shown in the following theorem.

Theorem 25 (Correctness of QG [MTN07]) *Let $\perp_e = h \leftarrow b_1, .., b_n$ be a bottom clause of example e with associated setting $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\perp \rangle$. The clause $F = qg(\perp_e, E)$ is in $Fringe(e, \mathcal{S})$.*

The following theorem considers whether every element of the fringe associated with a given example can be generated by the QG algorithm.

Theorem 26 (Completeness of QG [MTN07]) *Let $\perp_e = h \leftarrow b_1, .., b_n$ be a bottom clause of example e with associated setting $\mathcal{S} = \langle \mathcal{B}, E, \mathcal{L}, \succeq_\perp \rangle$. For each $F \in$*

[3] The negative-based reduction was first introduced in Golem [MF90] .

1. eastbound(A) :- has_car(A,B), infront(A,B), has_car(A,C), has_car(A,D), has_car(A,E), load(D,circle,1), long(B), short(D), infront(C,E), shape(B,rectangle), load(E,hexagon,1), closed(C), wheels(B,2), open(B), short(C), long(E), infront(B,C), wheels(C,2), shape(D,rectangle), open(D), infront(E,D), shape(E,rectangle), wheels(E,3), load(B, rectangle,3), wheels(D,2), open(E), load(C,triangle,1), shape(C,rectangle).
2. eastbound(A) :- has_car(A,B), load(B,hexagon,1), has_car(A,C), infront(A,C), has_car(A,D), has_car(A,E), load(E,circle,1), long(C), short(E), infront(D,B), shape(C,rectangle).
3. eastbound(A) :- has_car(A,B), load(B,hexagon,1).

Figure 7.10: An example of the iterative reduction of a clause by the QG algorithm.

Fringe$(e, \mathcal{S})$ *there exists a stochastic derivation such that* $F = qg(\bot_e, E)$.

The following theorem considers the complexity of QG algorithm.

Theorem 27 (Complexity of QG) *An upper-bound on the time complexity for QG algorithm is* $O(|C|.|\bot_e|.|E^-|.c_r)$*, where* $|C|$ *is the cardinality of the reduced clause* C*,* $|\bot_e|$ *is the cardinality of the bottom clause,* $|E^-|$ *the cardinality of negative examples and* c_r *is a constant representing the maximum cost of theorem proving with a resolutions bound of* r.

Proof. By construction each atom b_j in the body of $C = h \leftarrow b_1, .., b_m$ was either a cutoff atom or an element of one of the minimal support sets in one of the cycles of the reduce algorithm. In the worst case all b_j are cutoff atoms and since the last atom is a cutoff atom twice the reduce algorithm returns C in at most $m + 1 = |C|$ cycles. The cost of finding a cutoff atom is proportional to length of the bottom clause, number of negative examples and the cost of theorem proving. The upper-bound on the cost of finding a cutoff atom can therefore be estimated by $|\bot_e|.|E^-|.c_r$. Hence, an upper-bound on the time complexity of QG algorithm is $O(|C|.|\bot_e|.|E^-|.c_r)$.

Figure 7.10 gives an illustration of the cycle-complexity result given in Theorem 27. As shown in this figure the reduced clause has 2 atoms in the body and the algorithm terminates after 2+1 cycles.

The QG algorithm described in this sections can be used for efficiently sampling from consistent clauses. Clauses generated by QG are consistent but not necessarily compressive with respect to the positive examples and they may have small or even no positive coverage. On the other hand, Progol-like cover-set algorithms generate hypotheses by adding clauses which provide a positive compression over training data (see Figure 7.1). A simple integration of QG in Progol can be realised by replacing Progol's A^* search by a QG sampling mechanism. This is implemented in GA-Progol as shown in Figure 7.1, i.e. when user-defined parameter *search_mode* is *QG*. This involves a program which returns a clause with highest positive compression from a sample of S calls to *qg* algorithm. In this setting, the program simply returns a consistent clause with the highest positive compression. According to Proposition 21, a sample size can be estimated based on the percentage of consistent clauses which have positive compression, so that at least one of the clauses has positive compression.

Clauses generated by the simple QG setting described above lack diversity and optimality. GA-Progol also implements a more advanced setting in which a GA search is used to evolve and re-combine clauses generated by QG. In this setting QG is used to seed a population of clauses processed by the GA. This setting can be selected when *search_mode* is set to QG/GA (see Figure 7.1). As described in Section 7.1.2, in a "GA only" setting and when the *encoding_mode* is set to *atoms_only*, the initial population of the GA consists of randomly generated bit-strings with length L, where L is the number of literals in the bottom clause. In the QG/GA setting, the initial population of the GA consists of clauses generated by the QG algorithm. As the QG algorithm generates clauses from the permutations of the same bottom clause, encoding these clauses into bit-strings is straightforward and follows the same scheme as shown in Figure 7.3.

7.3 ProGolem

As shown in Section 7.1.1, an efficient operator can be implemented for least generalisation in the subsumption order relative to a bottom clause (i.e. $lgg_\perp$). Operator $lgg_\perp$ is defined for a lattice bounded by a bottom clause $\perp_e$ which is constructed with respect to a single positive example e. In this section we describe a variant of Plotkin's RLGG, called asymmetric relative minimal generalisation (ARMG) or $armg_\perp$ which is based on subsumption order relative to a bottom clause $\bot_e$. Unlike $lgg_\bot$, $armg_\bot$ is defined with respect to a pair of positive examples. An ARMG is constructed iteratively from positive examples and as in Golem, by construction it is guaranteed to cover all positive examples which are used to construct it. However, as for $lgg_\bot$ the asymmetric relative minimal generalisation described in this section does not need the determinacy restrictions which was needed in Golem (see Section 7.1.3). Hence, ARMGs have the same advantage as RLGGs in Golem but unlike RLGGs the length of ARMGs is bounded by the length of $\bot_e$. In this section we describe the algorithm for constructing ARMGs which has been implemented in ProGolem. ProGolem is an ILP system which combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. ProGolem was first described in [MSTN10a] and has been implemented by Jose Santos as part of the General Inductive Logic Programming System (GILPS) [4].

7.3.1 ARMG algorithm

The asymmetric relative minimal generalisation of examples e' and e relative to $\bot_e$ is denoted by $armg_\bot(e'|e)$ and in general $armg_\bot(e'|e) \neq armg_\bot(e|e')$. In the following, first we define asymmetric relative minimal generalisation and study some of its properties and then we give an algorithm for constructing ARMG.

Definition 95 (Asymmetric relative common generalisation) *Let E, B and $\bot_e$ be as defined in Definition 43, $\overrightarrow{\mathcal{L}}_\bot$ as defined in Definition 53, e and e' be positive examples in E and $\overrightarrow{C}$ is a head-connected definite ordered clause in $\overrightarrow{\mathcal{L}}_\bot$. $\overrightarrow{C}$ is an asymmetric common generalisation of e' and e relative to $\bot_e$, denoted by $\overrightarrow{C} \in arcg_\bot(e'|e)$, if $\overrightarrow{C} \succeq_\bot \bot_e$ and $B \wedge C \vdash e'$*

Example 48 *Let $M = \{p(+), q(+,-), r(+,-)\}$ be mode definition, $B = \{q(a,a), r(a,a), q(b,b), q(b,c), r(c,d)\}$ be background knowledge and $e = p(a)$ and $e' = p(b)$ be positive examples. Then we have $\bot_e = p(X) \leftarrow q(X,X), r(X,X)$ and clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1,V_1)$, $\overrightarrow{D} = p(V_1) \leftarrow q(V_1,V_3), r(V_3,V_5)$ and $\overrightarrow{E} = p(V_1) \leftarrow q(V_1,V_3)$ are all in $arcg_\bot(e'|e)$.* $\diamond$

[4] Available from http://ilp.doc.ic.ac.uk/GILPS/

Definition 96 (Asymmetric relative minimal generalisation) *Let E and $\perp_e$ be as defined in Definition 43, e and e' be positive examples in E and $arcg_{\perp}(e'|e)$ be as defined in Definition 95. $\overrightarrow{C}$ is an asymmetric minimal generalisation of e' and e relative to $\perp_e$, denoted by $\overrightarrow{C} \in armg_{\perp}(e'|e)$, if $\overrightarrow{C} \in arcg_{\perp}(e'|e)$ and $\overrightarrow{C} \succeq_{\perp} \overrightarrow{C'} \in arcg_{\perp}(e'|e)$ implies $\overrightarrow{C}$ is subsumption-equivalent to $\overrightarrow{C'}$ relative to $\perp_e$.*

Example 49 *Let B, $\perp_e$, e and e' be as in Example 48. Then clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1, V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1, V_3), r(V_3, V_5)$ are both in $armg_{\perp}(e'|e)$.* $\diamond$

The following proposition shows that ARMGs are not unique.

Proposition 26 *The set $armg_{\perp}(e'|e)$ can contain more than one clause which are not subsumption-equivalent relative to $\perp_e$.*

Proof. In Example 48, clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1, V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1, V_3), r(V_3, V_5)$ are both in $armg_{\perp}(e'|e)$ but not subsumption-equivalent relative to $\perp_e$. $\square$

The following theorem shows that the length of ARMG is bounded by the length of $\perp_e$.

Theorem 28 *For each $\overrightarrow{C} \in armg_{\perp}(e'|e)$ the length of $\overrightarrow{C}$ is bounded by the length of $\perp_e$.*

Proof. Let $\overrightarrow{C} \in armg_{\perp}(e'|e)$. Then by definition there exist a substitution θ such that $\overrightarrow{C}\theta$ is a subsequence of $\perp_e$. Hence, the length of $\overrightarrow{C}$ is bounded by the length of $\perp_e$. $\square$

It follows from Theorem 28 that the number of literals in an ARMG is bounded by the length of $\perp_e$, which according to Proposition 23 is polynomially bounded in the number of mode declarations for fixed values of i and j, where i is the maximum variable depth and j is the maximum arity of any predicate in M. Hence, unlike the RLGGs used in Golem, ARMGs do not need the determinacy restrictions and can be used in a wider range of problems including those which are non-determinate. In the following we show that there is also an efficient algorithm for constructing ARMGs. The following definitions are used to describe the ARMG algorithm.

armg - ARMG construction algorithm

```
Input: Clause $\overrightarrow{B}$, Positive example $e'$
Output: ARMG clause $\overrightarrow{C}$ constructed from $\overrightarrow{B}$ and $e'$
01 Let $\overrightarrow{C}$ be $\overrightarrow{B} = h \leftarrow b_1, .., b_n$
02 while there is a blocking atom $b_i$ wrt $e'$ in the body of $\overrightarrow{C}$
03   remove $b_i$ from $\overrightarrow{C}$
04   remove atoms from $\overrightarrow{C}$ which are not head-connected
05 end while
06 return $\overrightarrow{C}$
```

Figure 7.11: Asymmetric Relative Minimal Generalisation (ARMG) algorithm.

Definition 97 (Blocking atom) *Let B be background knowledge, E^+ the set of positive examples, $e \in E^+$ and $\overrightarrow{C} = h \leftarrow b_1, \ldots, b_n$ be a definite ordered clause. b_i is a blocking atom if and only if i is the least value such that for all θ, $e = h\theta, B \nvdash (b_1, \ldots, b_i)\theta$.*

An algorithm for constructing ARMGs is given in Figure 7.11. Given the bottom clause $\perp_e$ associated with a particular positive example e, this algorithm works by dropping a minimal set of atoms from the body to allow coverage of a second example. Below we prove the correctness of the ARMG algorithm.

Theorem 29 (Correctness of ARMG algorithm) *Let E and $\perp_e$ be as defined in Definition 43, e and e' be positive examples in E, $armg_\perp(e'|e)$ be as defined in Definition 96 and $armg(\perp_e, e')$ be the algorithm given in Figure 7.11. Then $\overrightarrow{C} = armg(\perp_e, e')$ is in $armg_\perp(e'|e)$.*

Proof. Assume $\overrightarrow{C} \notin armg_\perp(e'|e)$. In this case, either $\overrightarrow{C}$ is not an asymmetric common generalisation of e and e' or it is not minimal. However, by construction $\overrightarrow{C}$ is a subsequence of $\perp_e$ in which all blocking literals with respect to e' are removed and therefore $B \wedge C \vdash e'$. Hence, $\overrightarrow{C}$ is an asymmetric common generalisation of e and e'. So, $\overrightarrow{C}$ must be non-minimal. If $\overrightarrow{C}$ is non-minimal then $\overrightarrow{C} \succeq_\perp \overrightarrow{C'}$ for $\overrightarrow{C'} \in armg_\perp(e'|e)$ which must either have literals not found in $\overrightarrow{C}$ or there is a substitution θ such that $\overrightarrow{C}\theta = \overrightarrow{C'}$. But we have deleted the minimal set of literals. This is a minimal set since leaving a blocking atom would mean $B \wedge C \nvdash e'$ and leaving a non-head-connected literal means $\overrightarrow{C} \notin armg_\perp(e'|e)$. So it must be the second case. However, in the second case θ must be a renaming since the literals in $\overrightarrow{C}$ are all from $\bot_e$. Hence, $\overrightarrow{C}$ and $\overrightarrow{C'}$ are variants and this contradicts the assumption and completes the proof. $\square$

The following example shows that the ARMGs algorithm is not complete.

Example 50 *Let B, $\bot_e$, e and e' be as in Example 48. Then clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1, V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1, V_3), r(V_3, V_5)$ are both in $armg_{\bot}(e'|e)$. However, the ARMGs algorithm given in Figure 7.11 cannot generate clause $\overrightarrow{D}$.* $\diamond$

Example 50 shows that the ARMGs algorithm does not consider hypotheses which require 'variable splitting'. As shown in Chapter 3 (Example 11), there is a group of problems which cannot be learned by a Progol-like ILP system without variable splitting. The concept of variable splitting and the ways it has been done in Progol and Aleph were discussed in Chapter 3. A partition-based representation and an implementation for variable splitting was also given in Section 7.1.1. Similar approaches could be adapted for ProGolem, however, the current implementation does not support variable splitting.

Figure 7.12 gives a comparison between Golem's determinate RLGG and the ARMGs generated by the ARMG algorithm in Michalski's east-bound trains problem. Note that Golem's RLGG cannot handle the predicate *has_car* because it is non-determinate (see Section 2.5). The first ARMG (2) subsumes the target concept which is eastbound(A) $\leftarrow$ has_car(A,B), closed(B), short(B). Note that in this example RLGG (1) is shorter than ARMGs (2,3) since it only contains determinate literals.

7.3.2 Search for best ARMG

ProGolem uses a cover set approach similar to the one used by Golem and Progol. ProGolem's cover set algorithm is shown in Figure 7.13. This algorithm repeatedly constructs a clause from a set of best ARMGs. As in Golem and QG/GA, ProGolem uses negative examples to reduce the final ARMGs before adding them to the current clausal theory H. The clause reduction algorithm implemented in ProGolem is similar to the reduce algorithm shown in Figure 7.9 but instead of a linear search used in Golem and QG/GA, it uses an adapted binary search to find the first blocking literal.

1. $rlgg(e_1, e_2) = rlgg(e_2, e_1)$ = eastbound(A) ← infront(A,B), short(B), open(B), shape(B,C), load(B,triangle,1), wheels(B,2), infront(B,D), shape(D, rectangle), load(D,E,1), wheels(D,F), infront(D,G), closed(G), short(G), shape(G,H), load(G,I,1), wheels(G,2).

2. $armg(\perp_{e_1}, e_2)$ = eastbound(A) ← has_car(A,B), has_car(A,C), has_car(A,D), has_car(A,E), infront(A,E), closed(C), short(B), short(C), short(D), short(E), open(B), open(D), open(E), shape(B,F), shape(C,G), shape(D,F), shape(E,H), load(D,I,J),2), wheels(E,2)

3. $armg(\perp_{e_2}, e_1)$ = eastbound(A) ← has_car(A,B), has_car(A,C), has_car(A,D), infront(A,D), closed(C), short(B), short(D), open(D), shape(B,E), shape(D,E), load(B,F,G), load(D,H,G), wheels(B,2), wheels(D,2)

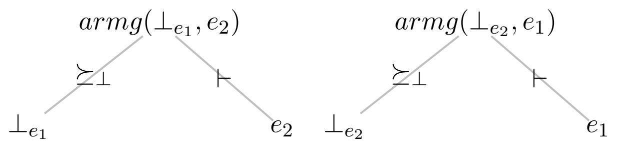


Figure 7.12: A comparison between Golem's determinate RLGG (1) and the non-determinate ARMGs (2,3). Note that Golem's RLGG cannot handle the predicate *has_car* because it is non-determinate. The first ARMG (2) subsumes the target concept which is eastbound(A) ← has_car(A,B), closed(B), short(B).

ProGolem's cover-set algorithm

```
Input: Examples $E$, mode declarations $M$, background knowledge $B$,
       $search\_mode$ ($beam/stochastic$)
Output: Hypotheses $H$
01 $H := \emptyset$
02 $E^+ :=$ positive examples in $E$
03 while $E^+ \neq \emptyset$
04   $e :=$ first example in $E^+$
05   Construct the bottom clause $\perp_e$ from $e$, $M$ and $B$ (See Appendix A)
06   $C' :=$ best_armg($\perp_e$, $E$, $search\_mode$) (See Fig. 7.14)
07   $C :=$ reduce($C'$, $E$) (See Fig. 7.9)
08   if compression($C$) > 0
09     $H := H \cup C$
10     $E^+_H :=$ positive examples covered by $B \wedge H$
11     $E^+ := E^+ - E^+_H$
12   end if
13 end while
14 return $H$
```

Figure 7.13: ProGolem's cover set algorithm.

best_armg - best ARMG algorithm

```
Input: $\perp_e$, Examples $E$, $search\_mode$ ($beam/stochastic$)
       sample size $K$, beam width $N$
Output: highest scoring clause from $best\_armgs$
01 let $best\_armgs = \{\perp_e\}$
02 repeat
03  let $best\_score$ = highest score from $best\_armgs$
04  let $Ex = K$ random positive examples from $E$
05  let $new\_armgs = \{\}$
06  for each $\overrightarrow{C} \in best\_armgs$ do
07   for each $e' \in Ex$ do
08     let $\overrightarrow{C'}$ = armg($\overrightarrow{C}$, $e'$) (see Fig. 7.11)
09     if $score(\overrightarrow{C'}) > best\_score$ then
10       $new\_armgs = new\_armgs \cup \overrightarrow{C'}$
11     end if
12   end for
13  end for
14  if ($new\_armgs \neq \{\}$) then
15   $best\_armgs$ = select_n($new\_armgs$, $N$, $search\_mode$)
16  end if
17 until $new\_armgs = \{\}$
```

select_n - select N highest scoring clauses

```
Input: set of clauses $S$, number of clauses to select $N$, $search\_mode$ ($beam/stochastic$)
Output: $N$ clauses from $S$ with highest scores
01 if $search\_mode = stochastic$ then
02  $S'$= clauses selected from $S$ by $N$ calls to tournament selection (see Fig. 7.7)
03 else
04  $S'$= $N$ highest scoring clauses from $S$
05 return $S'$
```

Figure 7.14: Best ARMG algorithm

ProGolem also implements several efficient coverage testing algorithms. Note that ProGolem is a bottom-up system and the coverage testing of long non-determinate clauses could be expensive as it involves a large amount of backtrackings. The details of ProGolem's efficient coverage testing algorithms are described in [San10].

In the following we describe ProGolem's search for best ARMG. ProGolem uses a beam search to select the best ARMG with respect to $\bot_e$. This algorithm is shown in Figure 7.14. The algorithm works by repeatedly constructing new ARMGs from current ARMGs using positive examples and maintaining a 'beam' of best ARMGs at each iteration to be used in the next iteration. Positive examples used to construct a new ARMG are randomly selected from the set of positive examples which have not been used before for that ARMG. This is repeated until the ARMGs' score no longer increases. ProGolem supports several evaluation functions (i.e. compression, accuracy, precision and coverage) which can be selected by the user. The default evaluation is compression as used in Progol and GA-Progol.

As shown in Figure 7.14, the initial set of ARMGs *best_armgs*, at iteration 0 is set to the bottom clause $\{\bot_e\}$. For each ARMG $\overrightarrow{C}$ in the current *best_armgs*, K (sample size) examples which are not covered by $\overrightarrow{C}$ are randomly selected. These examples are used to construct K new ARMGs using the algorithm shown in Figure 7.11, and from these new ARMGs, those with a score higher than *best_score* from the previous iteration are added to *new_armgs*. The best N (beam width) ARMGs from *new_armgs* are selected as *best_armgs* using *select_n*.

As shown in Figure 7.14, *select_n* returns N highest scoring clauses from a set of clauses S. However, if *stochastic_beam* is set to true, clauses are selected from S by N calls to the tournament selection (see Fig. 7.7). This is used to implement a GA-like algorithm in which clauses for ARMGs are stochastically selected from previous generation using a tournament selection. ProGolem also implements a stochastic search based on QG (see Section 7.2) in which bottom clauses are only reduced using QG instead of being passed to the ARMG algorithm.

7.4 Related work and discussion

The genetic algorithm approach described in this chapter is related to relational learning systems which use genetic algorithms for learning first-order clauses from examples. In particular, our GA approach can be compared with GA-based approaches which use a (user defined) language template for mapping first-order clauses into bit strings, i.e. GA-SMART [GS92], REGAL [GN96], DOGMA [Hek98] and G-NET [AGLS98]. As discussed in Section 6.3, the main problem of using these templates is that the number of conjuncts grows combinatorially with the number of predicates and also the template should be defined by the user. We also showed that these systems cannot benefit from intentional background knowledge in the same way as in an ILP system.

On the other hand, in our proposed framework, encoding of hypotheses is based on a most specific (or bottom) clause which is constructed according to the background knowledge. This bottom-clause can be automatically constructed using logic-based methods such as Inverse Entailment. Moreover, it was shown that the binary representation and operators described in this chapter encode the partition lattice which is isomorphic with the bounded subsumption. In other words, the proposed encoding and operators can be interpreted in well known ILP terms which means it follows *the principle of meaningful building blocks* [Gol89].

As already mentioned in this chapter, ProGolem is closely related to Golem which is based on generalisation relative to background knowledge B. ProGolem is based on generalisation relative to a bottom clause $\bot_e$ which is the result of compiling background knowledge B. Hence, subsumption relative to a bottom clause can be viewed as subsumption relative to a compilation of B which makes it more efficient than subsumption relative to B. Moreover, as already discussed in this thesis, generalisation relative to a bottom clause allows ProGolem to be used for non-determinate problems where Golem is inapplicable.

The least and minimal generalisations relative to a bottom clause (i.e. $lgg_\bot$ and $armg_\bot$) can be compared with other approaches which use lgg-like operators but instead of considering all pairs of compatible literals they only consider one pair. For example, LOGAN-H [AK04] is a bottom-up system which is based on inner products of examples which are closely related to lgg operator. This system constructs lgg-like clauses by considering only those pairs of literals which guarantee an injective mapping between variables. In other words, it assumes one-one object mappings. Other similar approaches use the same idea of simplifying the lgg-like operations by considering only one pair of compatible literals but they select this pair arbitrarily (e.g. [BZB01]).

7.5 Summary

In this chapter we described several algorithms and implementations for searching the hypothesis space bounded by a bottom clause. In the first part we described GA-Progol, an extension to the ILP system Progol in which the standard refinement and the A^*-like algorithm for searching the bounded subsumption lattice is replaced by a genetic search. We described the SGA-like algorithm in GA-Progol which uses a novel binary representation and encoding for clauses in the bounded subsumption lattice. The binary representation encodes the partition lattice which is in turn isomorphic with the bounded subsumption as discussed in the previous chapters. We also showed that lgg(least general generalisation) and mgs(most general specialisation) for the bounded subsumption lattice, i.e. $lgg_\perp$ and $mgs_\perp$, can be implemented by simple bitwise operations on the binary encoding of clauses.

In this chapter we also described Quick Generalisation (QG) algorithm and QG/GA search which are implemented in GA-Progol. QG algorithm is a stochastic algorithm which constructs maximally general consistent clauses by randomly pruning Progol bottom clauses. The QG sampling algorithm can be viewed as a stochastic refinement search with unary refinement operator which randomly samples from "fringe" clauses (i.e. maximally general consistent clauses in the hypothesis space). We described a sampling algorithm based on QG which returns the clause with highest positive compression from a sample of s calls to QG. We also described a combination of QG and GA (i.e. QG/GA algorithm) in which the initial population of the GA consists of clauses generated by the QG algorithm.

We also described algorithms based on Asymmetric Relative Minimal Generalisation (ARMG) which are implemented in ProGolem. ProGolem combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. ARMG or $armg_{\perp}$ is defined based on subsumption relative to $\perp$ where the clause length is bounded by the length of the initial bottom clause. ProGolem, therefore do not need the determinacy restrictions used in Golem. In this section we discussed algorithms for constructing and searching ARMGs.

Chapter 8

Empirical evaluation

In this chapter we empirically evaluate the algorithms which were described in the previous chapter. This chapter is organised as a series of experiments. In each experiment we try to answer one or two questions about the performance of different algorithms. These questions are written as negative statements which we refer to as null hypotheses and each experiment is designed to refute or reject the null hypotheses. The reason why we use negative statements (null hypotheses) is that we only need to find one case which disproves a negative statement whereas proving a positive statement is usually not possible by performing experiments, even if the statement is shown to be true in many cases.

In the first experiment (Section 8.1.1), we study the convergence of the genetic search in GA-Progol and test if it can converge to an optimal solution on a simple ILP problem. We also test if using $lgg_\perp$ crossover operator can improve the convergence of the genetic search. In Section 8.1.2, we compare the performance of the GA search versus the A^*-like search in learning randomly generated concepts with different complexities. In Section 8.1.3, we compare the performance of GA and A^* on a typical ILP dataset with relatively short target clauses, i.e. mutagenesis problem. In Section 8.1.4, we evaluate GA and A^* on a set of problems involving long target clauses with different sizes, i.e. a subset of Phase Transition (PT) dataset. In Section 8.1.5, we demonstrate that GA-Progol can find the correct solution for some special cases where the solution cannot be found by Progol's refinement operator due to its incompleteness. In Sections 8.2.1 and 8.2.2, we examine QG/GA algorithm on mutagenesis and PT datasets respectively. In

Section 8.3, we empirically evaluate stochastic refinement in ProGolem (i.e. $armg_{\perp}$).

8.1 GA-Progol

8.1.1 Convergence of the genetic search

In this experiment we study the convergence of the genetic search in GA-Progol and test if it can converge to an optimal solution on a simple ILP problem. We also test if using $lgg_{\perp}$ crossover operator can improve the convergence of the genetic search. In this section we examine the following two null hypotheses:

Null hypothesis 1 *On a simple ILP problem, the genetic search in GA-Progol does not converge to an optimal solution.*

Null hypothesis 2 *Using $lgg_{\perp}$ crossover operator does not improve the convergence of the genetic search.*

Material and methods

In this experiment we used the SGA-like setting in GA-Progol, as described in Section 7.1.1, to learn Michalski's east-bound trains [Mic80]. The following parameter setting was used for SGA: $popsize = 30$, $p_m = 0.0333$ and $p_c = 0.6$. In this experiment we also compare the performance of SGA with $SGA + lgg_{\perp}$ which uses the task-specific operators $lgg_{\perp}$ as described in Section 7.1.3. The following parameter setting was used for $SGA + lgg_{\perp}$: $popsize = 30$, $p_m = 0.0333$, $p_c = 1 - \alpha * f$ and $p_{lgg} = \alpha * f$ where f is the mean value of the fitness of the parental strings ($f = \frac{f(s_1)+f(s_2)}{2}$) and $\alpha = 0.8$. The parameter setting for p_c and p_{lgg} is adapted from the parameter setting for generalisation crossover (p_g) used in GA-SMART [GS92] as described in Appendix B. The evaluation function used both SGA and $SGA + lgg_{\perp}$ is a function based on the compression value as used in the evaluation function of A^*-like search of Progol. However, the compression value is normalised [1] to be a value between 0 and 1 as defined below:

$$f(C) = \alpha \frac{e^{+}(C)}{(E^{+} + \beta * e^{-}(C))} + (1 - \alpha)(1 - \frac{c(C) + h(C)}{c_{max} + h_{max}}) \tag{8.1}$$

[1] The normalisation is similar to the one used in GA-SMART (see Appendix B).

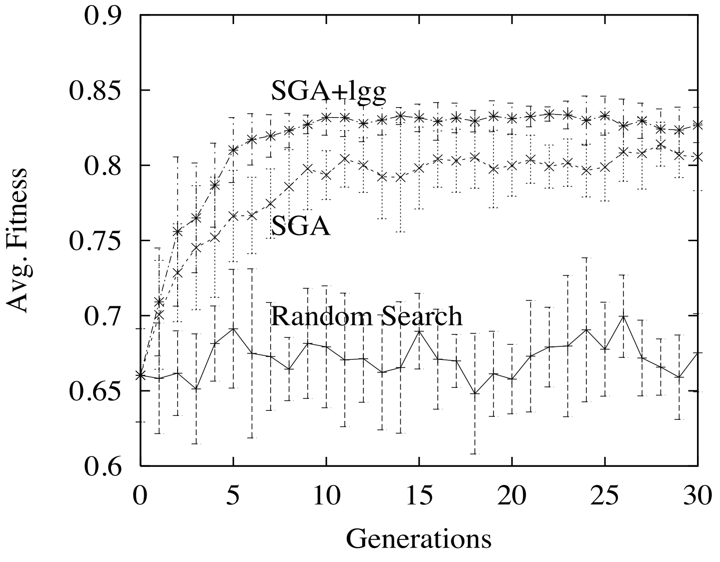


Figure 8.1: Convergence of the genetic search in the trains problem.

where E^+, E^- are total numbers of positive and negative examples, $e^+(C), e^-(C)$ are numbers of positive and negative examples covered by clause C, $c(C)$ is the length of clause C, $h(C)$ is the number of further literals to complete clause C as used in Progol (see Appendix A). Constant values c_{max} and h_{max} are maximum values for $c(C)$ and $h(C)$ when $C = \perp$, α and β are user-defined parameters, where α controls the effect of the clause coverage versus the effect of the clause length and β controls the effect of negative coverage versus the effect of positive coverage in the fitness function. In all experiments we used the following values for α and β: $\alpha = 0.8$ and $\beta = 0.5$. In this experiment *encoding_mode* is set to *var_splitting* (Figure 7.6) and selection method is set to roulette-wheel algorithm (Figure 7.7).

Results and discussion

Figure 8.1 compares the convergence of *SGA*, *SGA*+$lgg_\perp$ and a random search. In the random search each population is generated randomly as for GA's initial population and a hypothesis with the highest fitness value is selected at each generation. This graph shows the average fitness value of the populations versus number of generations. Standard deviations for the average fitness mean over 10 runs are shown as error bars.

```
carriage(Shape,Length,Double,Roof,Wheels,Load) :-
        shape(Length,Shape),
        double(Length,Shape,Double),
        roof(Length,Shape,Roof),
        wheels(Length,Wheels),
        load(Length,Load).
shape(long,rectangle). shape(short,rectangle). shape(short,ellipse). shape(short,hexagon).
shape(short,u_shaped). shape(short,bucket). double(short,rectangle,double).
double(long,rectangle,not_double). double(short,rectangle,not_double).
double(short,ellipse,not_double). double(short,hexagon,not_double).
double(short,u_shaped,not_double). double(short,bucket,not_double). roof(short,ellipse,arc).
roof(short,hexagon,flat). roof(long,rectangle,none). roof(long,rectangle,flat).
roof(long,rectangle,jagged). roof(short,rectangle,none). roof(short,rectangle,flat).
roof(short,rectangle,peaked). roof(short,u_shaped,none). roof(short,u_shaped,flat).
roof(short,u_shaped,peaked). roof(short,bucket,none). roof(short,bucket,flat).
roof(short,bucket,peaked). wheels(short,2). wheels(long,2). wheels(long,3).
load(short,l(circle,1)). load(short,l(diamond,1)). load(short,l(hexagon,1)).
load(short,l(rectangle,1)). load(short,l(triangle,1)). load(short,l(utriangle,1)).
load(short,l(circle,2)). load(short,l(diamond,2)). load(short,l(hexagon,2)).
load(short,l(rectangle,2)). load(short,l(triangle,2)). load(short,l(utriangle,2)).
load(long,l(circle,1)). load(long,l(diamond,1)). load(long,l(hexagon,1)).
load(long,l(rectangle,1)). load(long,l(triangle,1)). load(long,l(utriangle,1)).
load(long,l(circle,2)). load(long,l(diamond,2)). load(long,l(hexagon,2)).
load(long,l(rectangle,2)). load(long,l(triangle,2)). load(long,l(utriangle,2)).
load(long,l(circle,3)). load(long,l(diamond,3)). load(long,l(hexagon,3)).
load(long,l(rectangle,3)). load(long,l(triangle,3)). load(long,l(utriangle,3)).
```

Figure 8.2: Concept description language for generating random trains. A random train is defined as a list of carriages sampled from this program using SLPs.

In all experiments (10 out of 10 runs) an optimal solution (i.e. shortest consistent clause which covers all positive examples) was found by the SGA search before generation 20. Null hypothesis 1 is therefore refuted. As shown in the graph $SGA + lgg_\perp$ has a faster convergence to highly fit populations. In all experiments (10 out of 10 runs) an optimal solutions was found by $SGA + lgg_\perp$ before generation 10. This refutes Null hypothesis 2.

The rapid convergence of the GA search in this experiment also appears to be consistent with Schema Theorem [Hol75] which was discussed in Section 6.3. This theorem suggests that short, low-order and above average schemata receive at least exponentially increasing number of trials in successive generations.

8.1.2 Randomly generated concepts

In this section we compare the performance of the GA search versus the A^*-like search in learning randomly generated concepts.The Michalski's trains problem used in the previous experiment requires relatively short target clause, i.e. three literals in the body. In this experiment we use a program which generates random Michalski-style trains. This is used to generate positive and negative examples for target clauses with varying sizes. In this section we examine the following two null hypotheses:

Null hypothesis 3 *The GA search cannot provide increased efficiency over Progol's standard A^* search in learning any randomly generated target concept.*

Null hypothesis 4 *The increased efficiency in Null hypothesis 3 cannot be achieved without substantial decrease of accuracy.*

Material and methods

In this experiment, we compare the performance of the A^*-like search and the genetic search in learning concepts with different complexities. For this purpose, we use a stochastic concept generator program in which the concept description language is determined by a Stochastic Logic Program(SLP) [Mug96]. We use this to generate random concepts with a given complexity as well as random training and test examples for each concept. In the present experiment, we use a concept description language similar to Michalski's trains problem used in the previous experiment. Figure 8.2 shows the train description language used in this experiment. In this experiment, the complexity of target concept is defined as the number of specific features which describe the target concept. This is an estimation of the maximum number of literals to describe the concept. For example, complexity of the target concept in a train with a specific carriage with five distinct features (shape, load, etc.) is five. We can use a SLP program to generate complex target concepts as well an arbitrary number of training and test examples for each concept. The original non-SLP version of the random train generator program was written by Stephen Muggleton [2]. This program randomly generates trains with a given length and then classifies them as positive or negative examples. However, the non-SLP version is inefficient for generating examples for arbitrary long concepts. Figure 8.3 shows the experimental method used in this experiment. In this experiment we measure the average performance of the genetic search and the A^* search on 100 different runs. In each run, target concepts with complexities (i.e. maximum target size) between 5 to 25 are generated. For each

[2] Available from: http://www.doc.ic.ac.uk/~shm/Software/GenerateTrains/

```
01 for $i = 1$ to 100 do
02   for $j = 5, 10, 15, 20, 25$ do
03     Generate a random target concept $T_{ij}$ with complexity $j$
04     Generate $2 \times 100$ random training and test examples for concept $T_{ij}$
05     Run GA-Progol on the training examples using the $A^*$ search
06       $N_{ij}$ =number of evaluations before finding hypothesis $C_{ij}$
07       $A_{ij}$ =predictive accuracy of $C_{ij}$ on the test examples
08     Run GA-Progol on the training examples using the $GA$ search
09       $N'_{ij}$ =number of evaluations before finding hypothesis $C'_{ij}$
10       $A'_{ij}$ =predictive accuracy of $C'_{ij}$ on the test examples
11   end
12 end
13 for $j = 5, 10, 15, 20, 25$ do
14   Plot average and standard error of $N_{ij}$ and $N'_{ij}$ versus $j$ ($i \in [1..100]$)
15 for $j = 5, 10, 15, 20, 25$ do
16   Plot average and standard error of $A_{ij}$ and $A'_{ij}$ versus $j$ ($i \in [1..100]$)
```

Figure 8.3: Experimental method for comparing GA versus A^* on randomly generated concept with different complexities.

target concept, a fixed number (i.e. 100) of examples are generated for training and testing. After generating random examples, we run GA-Progol on the training example using the A^* search and the genetic search. For each iteration of the loop the following parameters are recorded: N, the number of evaluations before finding a single clause C which is complete and consistent with respect to the training examples and A, the predictive accuracy of C on the test examples. The average and standard error of these parameters are then plotted against the complexity of the target concepts.

Predictive accuracy is defined as the proportion of correctly predicted unseen test examples. Each clause is learned from a set of 100 randomly generated training examples and tested on a different set of 100 randomly generated test examples. We follow the recommendation in Progol's manual [3] about the choice of test strategy (for evaluating predictive accuracy) according to the number of examples. Given 100 training examples and 100 test examples we use a hold-out test strategy implemented in Progol's built-in predicate *test*/1. The evaluation function and control parameters which are used by the genetic search are the same as those used in Section 8.1.1.

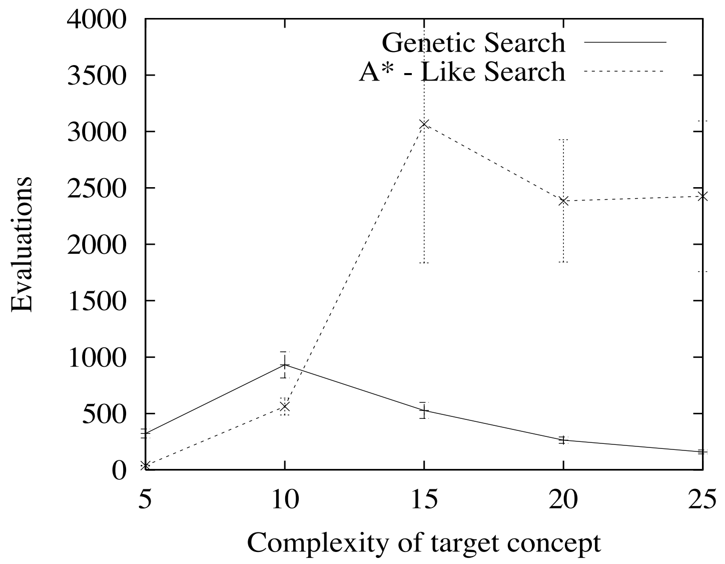
[3] Available from: http://www.doc.ic.ac.uk/~shm/Software/progol4.2/manual.ps

Results and discussion

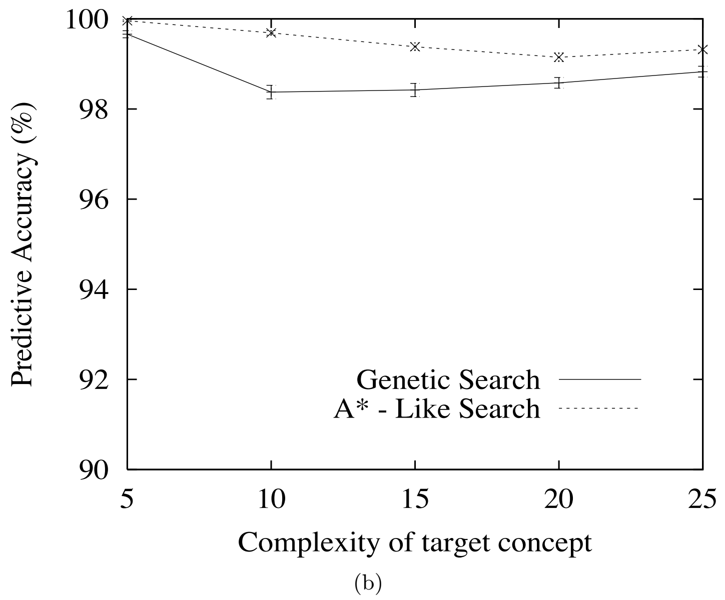
Figure 8.4.a compares the average number of clauses evaluated by the genetic search and the A^*-like search in learning concepts with different complexities. The vertical axis shows the average number of the explored nodes before finding a complete and consistent hypothesis. The horizontal axis shows the complexity of target concept which is estimated by the maximum size of the target clause. According to Figure 8.4, in this experiment the A^*-like search exhibits a better performance than GA search in learning concepts with small complexities, i.e. for complexities between 5 to 10 the GA search requires a higher number of evaluations compared to A^*-like search. However, the number of evaluations by the A^*-like search grows very sharply for complexities more than 10 and it is significantly higher than the number of evaluations by the GA search. The Null hypothesis 3 is therefore rejected.

Figure 8.4.b compares the predictive accuracy of the hypotheses by the genetic search and the A^*-like search. This figure shows that the overall predictive accuracy of the genetic search is lower than the overall predictive accuracy of the A^*-like search. However, the difference between the predictive accuracies in most cases is less than 1% and in all cases less than 2%. This rejects Null hypothesis 4. Slightly lower predictive accuracy of the genetic search can be explained by the fact that, unlike the A^*-like search, the genetic search does not guarantee to find an optimal clause, i.e. a clause with the highest compression. Moreover, a comparison between the size of hypotheses suggest that the clauses generated by the GA search are usually longer than the corresponding clauses generated by the A^*-like search. These over-specific clauses could contribute to the lower predictive accuracy of the GA search. In Sections 8.2 we examine a combination of the GA search with QG algorithm which can be used to prune over-specific clauses.

In summary, the results of this experiment suggest that in the random trains problem, the genetic search can lead to significantly increased efficiency for learning complex target concepts and this can be achieved without substantial decrease of accuracy. These graphs also suggest that the performance of the genetic search is less dependent on the complexity of hypotheses, whereas A^*-like search shows a great dependency on



(a)



(b)

Figure 8.4: Performance of *GA* search and *A**-like search in learning randomly generated concepts with different complexities. a) average number of clauses evaluated by each search method and b) predictive accuracy of the induced hypotheses versus the complexity of concepts.

this factor. This can partly be explained by the fact that in the GA search we always evaluate a fixed number of clauses (i.e. *popsize*) in each generation regardless of the length of the target clause, whereas the number of nodes which A^*-like search requires to explore and evaluate increases exponentially with the length of the target clause (see Section 7.1).

The graphs in Figure 8.4 also suggest that in this experiment, the most difficult problems are not necessarily those with longer target concepts as both for the GA and A^*-like search the peaks of computational costs are for target clauses with complexities somewhere between 10 to 15. This could be related to phase transition phenomena in relational learning [BGSS03], however, this requires further investigation which is beyond the scope of the current experiment. Nevertheless, in order to avoid this potential side effect, in Section 8.1.4 we evaluate GA-Progol on problems which are known to be outside the phase transition region.

8.1.3 Learning short target clauses

The goal of this experiment is to compare the performance of GA and A^* on a typical ILP benchmark problem involving short target clauses, i.e. target clauses with a small number of literals in the body. In this experiment we examine the following null hypothesis:

Null hypothesis 5 *The GA search cannot provide increased accuracy over Progol's standard A^* search on a typical ILP problem involving short target clauses.*

Material and methods

In this experiment we use the mutagenesis 42 dataset with atom-bond background knowledge [KMSS96], referred to as *mut*42. Mutagenesis problem has been widely used as a benchmark problem in the ILP community. Even though this problem is no longer regarded as a challenging problem for ILP, it is a good example of successful application of ILP involving short target clauses, i.e. up to 4 literals as discussed in Section 1.1.2. We use a leave-one-out test strategy and compare the predictive accuracy and total time (training and testing) for the following algorithms:

A^* - Progol's standard refinement-graph search.

GA - A Genetic Algorithm (GA) described as "SGA-like algorithm" in Section 7.1.1.

In this experiment N, the maximum number of clauses evaluated for learning a single clause, is varied for each algorithm. For A^*, N corresponds to Progol's parameter *nodes*. In GA, N is determined by *popsize* and *maxgen*, i.e. $N = popsize \times (maxgen + 1)$. Other GA parameters used in this experiment are as follows: $p_m = 0.01$, $p_c = 0.6$ and $popsize = 30$. For A^*, the maximum number of literals in each clause (parameter c in Progol) is set to 4. The *noise* parameter was set to zero for all algorithms. In this experiment *encoding_mode* is set to *atoms_only* (Figure 7.6) as the variable splitting is not needed for this problem and selection method is set to tournament selection algorithm (Figure 7.7) with a tournament size of 2. The evaluation function used for the GA search is the same as the one used by A^* search, i.e. compression value for each clause.

Results and discussion

Table 8.1 summarises the results of the leave-one-out experiments. This table shows predictive accuracies and average learning and testing times for each example. In these experiments, the value for N starts from 30 (for GA this is equivalent to $maxgen = 1$) and it is doubled each time up to 15360 ($maxgen = 512$). According to the table, predictive accuracies for both A^* and GA increase with N. In general, GA seems to reach higher levels of accuracy with fewer evaluated clauses. However, as the computational costs for each search node is relatively low in this problem (e.g. clauses considered by A^* are maximum 4 literals long), the efficiency advantage of GA is not significant in this problem. Given the relatively high accuracy errors in the leave-one-out experiments, the accuracy advantage of GA is also not significant. Null hypothesis 5 is therefore not rejected by this experiment. Nevertheless, the results from this experiment suggest that the predictive accuracies and timings for A^* and GA on mutagenesis problem (a benchmark problem involving short clauses) are comparable. These results also suggest that GA may reach higher accuracy with fewer evaluated clauses and therefore in a learning setting where the total computational cost is dominated by the cost of

N	A^*		GA	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
30	81 ± 6	0.11	83 ± 6	2.82
60	81 ± 6	0.18	83 ± 6	2.83
120	81 ± 6	0.41	81 ± 6	5.50
240	81 ± 6	1.09	81 ± 6	8.94
480	81 ± 6	1.84	83 ± 6	13.40
960	81 ± 6	4.27	83 ± 6	25.85
1920	81 ± 6	9.87	83 ± 6	43.46
3840	81 ± 6	25.10	83 ± 6	58.49
7680	83 ± 6	50.76	86 ± 5	99.98
15360	86 ± 5	141.06	86 ± 5	226.78

Table 8.1: Predictive accuracies and average learning times from leave-one-out experiments on mut42 dataset for GA vs A^*. N is the maximum number of clauses evaluated for learning a single clause.

evaluations (e.g. when learning long target clauses), then GA could lead to increased efficiency over Progol's standard A^* search. This is consistent with the results from random trains and is the basis of the experimental setting which will be used in the next section.

8.1.4 Learning long target clauses

The results of the previous experiment (i.e. Table 8.1) suggest that GA needs fewer clause evaluations in order to reach the same level of accuracy. For example, A^* requires $N = 7680$ to reach accuracy of 83 ± 6 and $N = 15360$ to reach accuracy of 86 ± 5 while GA reaches these levels of accuracy at $N = 480$ and $N = 7680$ respectively. This observation suggest that GA could lead to a significant speed up over A^* in cases where the average computational costs for each search node is relatively high, e.g. when the clauses which need to be considered during the search are relatively long. In this experiment we evaluate GA and A^* on a set of problems involving long clauses with different target sizes. In particular we examine the following null hypothesis:

Null hypothesis 6 *The GA search cannot provide increased accuracy over Progol's standard A^* search on an ILP problem involving long target clauses.*

Material and methods

In this experiment we use a set of benchmark problems with varying concept sizes from 6 to 16. These problems are selected from the Phase Transition study [BGSS03]. As

m	A^*		GA	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
6	98	3.22	99.5	5.83
7	99.5	633.16	99.5	12.99
8	100	1416.55	100	13.92
10	97.5	25852.80	95.5	74.68
11	80	37593.20	99	30.37
14	50	128314.00	79.5	529.67
16	59	55687.44	74	297.93

Table 8.2: Predictive accuracies and learning times for GA vs A^* on a set of learning problems with varying concept sizes from 6 to 16.

discussed in Section 8.1.2, in order to avoid the potential side effect of phase transition, we can evaluate GA-Progol on problems which are known to be outside the phase transition region. In this experiment we selected problems $m6.l12$ to $m16.l12$ from the first row of the (m, L) plane so that they only approach the phase transition region. Each problem includes 100 training and 100 test examples. We use a hold-out test strategy and compare the performance of A^* and GA. The GA parameters used in this experiment are the same as the ones used in the previous experiment except that *maxgen* is not varied and it is set to 20 for all experiments. The A^* parameter *nodes* is also fixed and set to 10000. The maximum number of literals for each clause (c) is also set to the concept size for the problem to be learned (i.e. 6 to 16).

Results and discussion

Table 8.2 shows predictive accuracies and average learning and testing times for A^* and GA. According to this table, for $m = 6$ the performances of A^* and GA are similar. However, for $m = 7$ to $m = 16$, GA has found a solution with a similar or better accuracy than A^* in significantly less time. Null hypothesis 6 is therefore rejected. These results also suggest that A^* could be more efficient than GA for small values of m (i.e. less than 6). However, this cannot be tested on the present dataset as the smallest values of m are either 5 or 6 depending on the value of l, and in the problem used in this experiment 6 is the smallest value.

8.1.5 Clauses which cannot be learned by Progol

In this section we demonstrate that GA-Progol can find the correct solution for some special cases where the solution is beyond the exploration power of Progol's refinement operator due to its incompleteness. In particular, we re-visit cases where a concept cannot be learned by Progol because of the choice of ordering in the bottom clause and the variable dependencies in the literals (see Section 3.2). As shown in Section 5.4, refinement operator ρ_2 which is based on injective subsumption relative to $\perp$, can address Progol's incompleteness which is due to the ordering of the literals (see Example 34). In this section we demonstrate that a refinement setting based on ρ_2 which is implemented in GA-Progol can be used to learn concepts which could not be learned by Progol. In particular we examine the following null hypothesis:

Null hypothesis 7 *GA-Progol cannot find hypotheses which are missed by Progol because of its incompleteness due to the ordering of literals.*

Material and methods

In this experiment we use a simple program similar to the one used in [BS99] to illustrate Progol's incompleteness. This program is shown in Figure 8.5. In this experiment Progol is used with default setting. For GA-Progol, *ga_lmode* is set to 2, indicating the language mode related to refinement operator ρ_2, i.e. $\overrightarrow{\mathcal{L}}^{i}_{\perp}$ as discussed in Section 7.1.2.

Results and discussion

For both Progol and GA-Progol, the following bottom clause is generated from the first example $p(1, a)$:

$$\perp : \mathrm{p}(A, B) \leftarrow \mathrm{f}(A, C), \mathrm{g}(A, B), \mathrm{h}(A, B)$$

```
% Mode declarations
:-modeh(1,p(+any,+t))?
:-modeb(1,f(+any,-t))?
:-modeb(1,g(+any,+t))?
:-modeb(1,h(+any,-t))?
% Positive examples
p(1,a).  p(2,a).  p(3,a).
f(1,b).  f(2,b).  f(3,b).  f(4,b).
g(1,a).  g(2,c).  g(3,c).
h(1,a).  h(2,c).  h(3,c).  h(4,a).
% Negative examples
:-p(4,a).
% Types
t(a).    t(b).    t(c).
```

Figure 8.5: A simple problem to illustrate Progol's incompleteness adapted from [BS99].

The following clauses are considered by Progol from the first example:

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathrm{p}(A,B).} \\ \mathrm{p}(A,B) & \leftarrow & \mathrm{f}(A,C). \\ \mathrm{p}(A,B) & \leftarrow & \mathrm{h}(A,C). \\ \mathrm{p}(A,B) & \leftarrow & \mathrm{f}(A,C), \mathrm{h}(A,D). \end{array}$$

None of the above clauses have positive compression and therefore Progol does not generate any hypotheses from the first example. Progol misses the correct solution $\mathrm{p}(A,B) \leftarrow \mathrm{g}(A,C), \mathrm{h}(A,C)$ because of the first form of incompleteness described in Section 3.2. This incompleteness is due to the choice of ordering in the bottom clause, the variable dependencies in the literals and the fact that Progol's refinement considers the literals in $\perp$ only from left to right. An alternative is to consider clauses in $\overrightarrow{\mathcal{L}}^i_\perp$ which are injective subset (rather than subsequence) of $\perp$ as discussed in Section 5.4. This can be activated in GA-Progol by setting *ga_lmode* to 2. GA-Progol can then find the correct solution from the first example. The null hypothesis 7 is therefore refuted. Note that even though Progol cannot find any solution from the first example, it can eventually find the correct solution from generalising the next training examples.

8.2 QG and QG/GA

8.2.1 Learning short target clauses

In this section we examine QG and QG/GA algorithms as described in Section 7.2. In particular we examine the following null hypothesis:

Null hypothesis 8 *A combination of QG and GA cannot provide increased accuracy over each individual approach on a typical ILP problem involving short target clauses.*

Material and methods

In this experiment we use the mut42 dataset from Section 8.1.3.

We use a leave-one-out test strategy and compare the predictive accuracy and total time (training and testing) for the following algorithms:

A^* - Progol's standard refinement-graph search.

QG - An implementation of the QG-sample algorithm as described in Section 7.2.

GA - A Genetic Algorithm (GA) described as "SGA-like algorithm" in Section 7.1.1.

QG/GA- A GA-based search in which QG has been used for seeding the GA population as described in Section 7.2.

In this experiment N, the maximum number of clauses evaluated for learning a single clause, is varied for each algorithm. For A^*, N corresponds to Progol's parameter *nodes*. In QG, N is the sample size. In GA and QG/GA, N is determined by *popsize* and *maxgen*, i.e. $N = popsize \times (maxgen + 1)$. Other GA and A^* parameters are the same as in Section 8.1.3.

Results and discussion

Table 8.3 summarises the results of the leave-one-out experiments. This table shows predictive accuracies and average learning and testing times for each example. According to the table, QG finds relatively good solutions right at the beginning when the

N	A^*		QG		GA		QG/GA	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
2	69 ± 7	0.12	76 ± 7	2.36				
5	69 ± 7	0.13	81 ± 6	4.90				
10	69 ± 7	0.13	86 ± 5	9.30				
20	76 ± 7	0.11	86 ± 5	20.79				
30	81 ± 6	0.11	86 ± 5	27.60	83 ± 6	2.82	83 ± 6	27.68
60	81 ± 6	0.18	86 ± 5	54.70	83 ± 6	2.83	83 ± 6	27.68
120	81 ± 6	0.41	83 ± 6	116.51	81 ± 6	5.50	81 ± 6	31.12
240	81 ± 6	1.09	86 ± 5	228.78	81 ± 6	8.94	81 ± 6	35.89
480	81 ± 6	1.84	86 ± 5	469.08	83 ± 6	13.40	85 ± 5	39.61
960	81 ± 6	4.27	83 ± 6	930.82	83 ± 6	25.85	83 ± 6	47.54
1920	81 ± 6	9.87	83 ± 6	1861.35	83 ± 6	43.46	86 ± 5	64.07
3840	81 ± 6	25.10	83 ± 6	3773.29	83 ± 6	58.49	88 ± 5	85.95
7680	83 ± 6	50.76	83 ± 6	7686.96	86 ± 5	99.98	88 ± 5	106.79
15360	86 ± 5	141.06	83 ± 6	13916.10	86 ± 5	226.78	88 ± 5	165.00

Table 8.3: Predictive accuracies and average learning times from leave-one-out experiments on mut42 dataset. N is the maximum number of clauses evaluated for learning a single clause.

sample size is small. However, the predictive accuracies have not been improved with increased sample size. The figures even suggest that the accuracies have been decreased possibly due to overfitting the training data. Unlike for QG, predictive accuracies for GA and QG/GA increase with N.

In general, it appears that QG/GA reaches the highest level of accuracy with fewer evaluated clauses (88 ± 5 in 85.95 seconds). However, as in Section 8.1.3, the efficiency and accuracy advantages of QG/GA are not significant in this problem due to relatively low computational costs and relatively high accuracy errors in the leave-one-out experiments. Null hypothesis 8 is therefore not rejected by this experiment. In the next experiment we examine *QG/GA* on a set of problems with long target clauses.

8.2.2 Learning long target clauses

In this experiment we evaluate *QG* and *QG/GA* on a set of problems involving long clauses with different target sizes as used in Section 8.1.4. In particular we examine the following null hypothesis:

Null hypothesis 9 *A combination of QG and GA cannot provide increased accuracy over each individual approach on an ILP problem involving long target clauses.*

m	A^*			QG		GA		QG/GA	
	$Cs(\%)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
6	31.71	98	3.22	99.5	3.89	99.5	5.83	99.5	10.32
7	3.36	99.5	633.16	99.5	45.11	99.5	12.99	99.5	86.51
8	1.05	100	1416.55	100	175.03	100	13.92	100	169.55
10	0.0015	97.5	25852.80	99	242.22	95.5	74.68	99	1064.22
11	0.36	80	37593.20	91	774.02	99	30.37	99.5	110.15
14	0	50	128314.00	69	4583.25	79.5	529.67	**88.5**	1184.76
16	4×10^{-6}	59	55687.44	77.5	4793.01	74	297.93	**89.5**	4945.20

Table 8.4: Predictive accuracies and learning times for different search algorithms on a set of learning problems with varying concept sizes from 6 to 16.

Material and methods

In this experiment we use the set of problems used in Section 8.1.4. Each problem includes 100 training and 100 test examples. We use a hold-out test strategy and compare the performance of the same algorithms used in Section 8.2.1 (i.e. A^*, QG, GA and QG/GA). The GA parameters used in this experiment are also the same as the ones used in Section 8.2.1 except that *maxgen* is not varied and it is set to 20 for all experiments. The A^* parameter *nodes* is fixed and set to 10000. The maximum number of literals for each clause (c) is also set to the concept size for the problem to be learned (i.e. 6 to 16).

Results and discussion

Table 8.4 shows predictive accuracies and average learning and testing times for different algorithms. According to this table, in all cases QG has found a solution with a similar or better accuracy than the A^* search in significantly less time. These results also suggest that we can get a better predictive accuracy by combining QG and GA, i.e. unlike A^* and QG, QG/GA passes the 80% accuracy criteria of [BGSS03] for $m = 14$ and $m = 16$. This rejects Null hypothesis 9. According to the table, the efficiency and accuracy advantages of GA and QG/GA are more evident in problems with long target clauses and when the density of consistent clauses ($Cs\%$) is low. Density of consistent clauses is taken as being the proportion of consistent clauses in the A^* search.

8.3 ARMGs and stochastic search in ProGolem

In this section we empirically evaluate Asymmetric Relative Minimal Generalisation (ARMG) relative to $\perp$ ($armg_{\perp}$) and compare stochastic searches in ProGolem as described in Section 7.3.1. In particular we examine the following null hypotheses:

Null hypothesis 10 *On our benchmark problems, ARMG cannot provide increased accuracy over QG.*

Null hypothesis 11 *On our benchmark problems, a GA-like search which uses a stochastic selection of best ARMGs cannot provide increased accuracy over a beam search.*

As described in Section 7.3.1, $armg_{\perp}$ in ProGolem can be viewed as a stochastic refinement operator relative to $\perp$. ProGolem also implements a stochastic search based on QG (see Section 7.2). Both $armg_{\perp}$ and QG are upward stochastic refinement operators relative to $\perp$ which work by pruning (i.e. reducing) bottom clauses. We evaluate QG and ARMG in ProGolem on several ILP datasets and also compare the results with a non-stochastic algorithm with unary refinement operator (i.e. Aleph). We also compare the standard beam search in ProGolem with a GA-like search in which clauses for ARMGs are stochastically selected from previous generation using a tournament selection.

Materials and methods

The datasets used in this experiment are the same as those used in the experiments in [MSTN10a], i.e. Alzheimers-Amine [KSS95], Carcinogenesis [SMS97], DSSTox [RW00], Metabolism [CHH^{+}02], Proteins [MKS92] and Pyrimidines [KMLS92]. In [MSTN10a], the performances of Golem, Aleph and ProGolem were compared on these datasets. In particular it was shown that while ProGolem has the advantages of Golem for learning large target clauses in Proteins and Pyrimidines datasets, it does not suffer from the determinacy limitation and can be used for other datasets where Golem is inapplicable. It was also shown that like Golem, ProGolem significantly outperforms Aleph on Proteins dataset while on other datasets shows a comparable performance with Aleph. In this

dataset	Aleph		QG		ARMG(beam search)		ARMG(GA-like search)	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
Alz-Amine	76.2±3.8	162	74.7±5.0	16	76.2±2.4	21	76.1±3.1	22
Carcino	59.7±6.3	58	52.0±6.9	8	63.9±6.3	23	65.9±7.3	23
DSSTox	72.6±6.9	239	64.1±10.0	74	64.7±9.0	74	64.7±9.0	74
Metabolism	62.1±6.2	32	59.7±11.2	2	67.6±13.6	13	66.7±13.5	18
Proteins	50.5	4502	49.3	243	64.9	8,648	62.5	1,834
Pyrimidines	73.7	23	75	107	74.1	174	**84.4**	180

Table 8.5: Predictive accuracies and learning times for stochastic searches in ProGolem.

section we compare the performance of QG and ARMG in ProGolem and we also evaluate a GA-like search in ProGolem which combines the beam search for best ARMGs with a tournament selection (see Figure 7.7). In order to use QG, $progolem\_mode$ is set to $reduce$ and for $ARMG$, $progolem\_refinement\_operator$ is set $armg$ and $progolem\_mode$ is set to $single$. For GA-like search, $progolem\_stochastic\_beam$ is set to true. This uses a tournament selection with tournament size equal to 2 as a default. We use the original hold-out test strategy used for Proteins [MKS92] and Pyrimidines [KMLS92], where respectively 4/5 and 2/3 of the data were used as training data and the remaining as the test data. For the Carcinogenesis, Metabolism and Alzheimers-Amine datasets, we use a 10-fold cross-validation and for DSSTox a 5-fold cross validation test strategy. In addition to average predictive accuracies, we also report standard deviations whenever cross-validation is used. Both Aleph and ProGolem were used with YAP Prolog (ver. 6) with the following parameter settings: $i = 2$, $maxneg = 30$ for Carcinogenesis and Proteins and $maxneg = 10$ for all other datasets, $evalfn = coverage$ for DSSTox and $evalfn = compression$ for all other datasets. Aleph was used with the following parameters: $nodes = 1000$ and $clauselength = 5$ for all datasets except for Proteins where $nodes = 10000$ and $clauselength = 40$. ProGolem was used with $N = 2$ (beam-width) and $K = 10$ (sample size at each iteration). As shown in Theorem 21, the sample size can be estimated as follows: assuming that the upper bound for the number of clauses in the target theory is 5 (for all problems), choosing a sample size of 2×5 means that the probability of generating a consistent $ARMG$ is at least $1 - e^{-2} = 0.86$

Results and discussion

Table 8.5 summarises the results of the experiments. According to the results, the predictive accuracies of *ARMG* are significantly higher than *QG* in several cases (e.g. Proteins dataset). Null hypothesis 10 is therefore rejected. The results also suggest that the GA-like search can outperform the beam search. The accuracy of *ARMG* with the GA-like search on Pyrimidines dataset (i.e. 84.4%) is significantly higher than Aleph, QG and *ARMG* with beam search and to our knowledge is the highest accuracy published to date for this dataset. This suggests a potential advantage of the GA-like search in ProGolem for some problems. Null hypothesis 11 is therefore rejected.

Both *QG* and *ARMG* are bottom-up algorithm and it is expected that they should have relatively better performance than Aleph in problems which require learning long target concepts. A better performance of *ARMG* (compared to *QG*) in this experiment could be explained by the fact that by construction *ARMGs* cover more and more positive examples while clauses generated by QG only guarantee to cover the positive example used to build the bottom clause. The apparent advantage of GA-like search over the beam search in Pyrimidines problem could be related to the advantage of the GA-like search for escaping from local optima due to a stochastic selection.

8.4 Summary

In this chapter we evaluated the implementations described earlier in this thesis. In the first experiment, we studied the convergence of the genetic search in GA-Progol and showed the convergence of the search to an optimal solution on a simple ILP problem, i.e. Michalski's east-bound trains problem. We also showed that using $lgg_{\perp}$ crossover operator can improve the convergence of the genetic search, i.e. an optimal solution can be found with fewer generations of the GA. In the second experiment we compared the performance of the *GA* search versus the A^*-like search in learning randomly generated concepts, i.e. random Michalski-style trains with different complexities. The results of this experiment suggest that in the random trains problem, the genetic search can lead to significantly increased efficiency for learning complex target concepts and this can be achieved without substantial decrease of accuracy. In the third experiment we compared the performance of GA and A^* on a typical ILP benchmark problem involving short target clauses, i.e. mutagenesis dataset. The results from this experiment suggested that the predictive accuracies and timings for A^* and GA on mutagenesis problem (a benchmark problem involving short clauses) are comparable. These results also suggested that GA may reach higher accuracy with fewer evaluated clauses. This was therefore consistent with the results from random trains and was the basis of the next experimental setting. In the fourth experiment we compared GA and A^* on a set of problems involving long target clauses with different sizes, i.e. a subset of Phase Transition (PT) dataset. The results clearly suggested the advantage of GA for learning long clauses, i.e. GA found a solution with a similar or better accuracy than A^* in significantly less time. In the fifth experiment we demonstrated that GA-Progol can find the correct solution for some special cases where the solution cannot be found by Progol's refinement operator due to its incompleteness. In the sixth and the seventh experiments we examined QG and QG/GA algorithms and tested if a combination of QG and GA can provide increased accuracy over each individual approach on i) mutagenesis and ii) PT datasets. The results indicated the efficiency and accuracy advantages of QG/GA in particular in problems with long target clauses and when the density of consistent clauses ($Cs\%$) is low. In the last experiment we evaluated ARMG and compared stochastic searches in ProGolem. We compared the performance of QG and ARMG in ProGolem on a set of problems. We also compared the standard beam search in ProGolem with a GA-like search in which clauses for ARMGs are stochastically selected from previous generation using a tournament selection. The results suggested a potential advantage of the GA-like search in some problems, i.e. the accuracy of $ARMG$ with the GA-like search on Pyrimidines dataset is significantly higher than Aleph, QG and $ARMG$ with beam search.

The overall conclusion from the experiments can be summarised as follows.

1. In ILP problems involving short target clauses (e.g. up to 6 literals), systematic search algorithms such as A^* can outperform stochastic searches such as GA and QG.

2. In ILP problems involving long target clauses (e.g. more than 6 literals), stochastic

searches such as GA and QG can outperform systematic search algorithms such as A^*.

3. A combination of QG and GA can provide increased accuracy over each individual approach

4. Using $lgg_\perp$ crossover operator can improve the convergence of the genetic search.

5. Using injective subsumption relative to $\perp$, GA-Progol can find hypotheses which are missed by Progol because of its incompleteness due to the ordering of literals.

6. In problems involving local optima a GA-like search in ProGolem can provide increased accuracy over a beam search.

Chapter 9

Conclusions

9.1 Summary of contributions

In this section we review the main contributions of this thesis. The main research questions and the overall contributions of the thesis were set out in Chapter 1. In this section we examine each contribution in more detail with respect to the research questions motivated in Chapter 1. We also describe whether and how each research question has been addressed.

1. **Characterising clause refinement within a bounded hypothesis space and the concepts of sequential subsumption and subsumption relative to a bottom clause**
   Several state of the art ILP systems (e.g. Progol and Aleph) are based on Inverse Entailment (IE) and use a special form of refinement to search through a hypothesis space bounded by a most specific (bottom) clause. In this thesis we gave a new analysis of refinement in this setting. In particular, clause refinement in Progol was revisited and it was demonstrated that Progol's refinement is incomplete with respect to the general subsumption order (i.e. θ-subsumption). Based on this analysis we introduced a subsumption order relative to a bottom clause and demonstrated how Progol's refinement can be characterised with respect to this order. This new subsumption order (referred to as bounded subsumption) is based on sequential subsumption which was also introduced in this thesis. We demonstrated that bounded subsumption order, unlike previously suggested orders, characterises all as-

pects of Progol's refinement. We have also shown that Progol's refinement operator is (weakly) complete with respect to the bounded subsumption order.

2. **The lattice and cover structure of bounded subsumption**

   We have shown that the most general specialisation and least general generalisation for the subsumption order relative to the bottom clause (i.e. $mgs_\perp$ and $lgg_\perp$) exist and therefore bounded subsumption forms a lattice. We have also defined downward covers for the bounded subsumption and have shown that, unlike for θ-subsumption, a finite set of downward covers exist.

3. **Morphisms of bounded subsumption**

   We have defined a mapping between the lattice of bounded subsumption and an atomic lattice and have shown that these two lattices are isomorphic. We have also shown that the atomic lattice is isomorphic to a lattice of partitions. Hence, the lattice of bounded subsumption can be encoded as a lattice of partitions and therefore clause refinement can be mapped to partition refinement.

4. **Encoding and ideal refinement operator for bounded subsumption and efficient $lgg_\perp$ and $armg_\perp$ operators**

   We have shown that, unlike for θ-subsumption, ideal refinement operators exist for bounded subsumption. We have defined a refinement operator ρ_1 for the lattice of bounded subsumption, i.e. $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. Each clause $\overrightarrow{C}$ in this lattice is encoded by a tuple $\langle K, \theta \rangle$. This encoding is based on the morphism between the lattice of bounded subsumption and the lattice of partitions mentioned above. The refinement operator ρ_1 works directly on the encoding tuples and uses a mapping function (c) which maps a tuple $\langle K, \theta \rangle$ into an ordered clause $\overrightarrow{C}$ in $\overrightarrow{\mathcal{L}}_\perp$. This refinement operator is the basis of the genetic operators which work on the encoding of the clauses in a GA-ILP search (see Chapter 7). We have proved that ρ_1 is ideal (finite, complete and proper). Note that, unlike in a systematic search which starts from a particular clause, in a stochastic search which could start from any point of the search space, the condition of weak completeness is not enough. Hence, in a stochastic search a complete refinement operator is usually preferred over a weakly complete operator even if this can lead to redundancy (a refinement operator cannot be both complete and non-redundant). We also studied the mapping and the morphism between $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and a lattice of the encoding tuples $\langle K, \theta \rangle$. We have shown that the mapping function c is a lattice isomorphism and lattices $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ and $\langle \mathcal{K} \times \Theta, \subseteq \rangle$ are two isomorphic lattices. As shown in Chapter 5, this isomorphism is important from a practical point of view and by contrast with the general subsumption order, efficient least generalisation operators ($lgg_\perp$ and $armg_\perp$) can be implemented for the bounded subsumption (see Chapter 7). We have shown that a variant of Plotkin's Relative RLGG which does not need the determinacy restrictions can be designed based on subsumption with respect to a bottom clause. This is the basis of Asymmetric Relative Minimal Generalisations (or ARMGs) relative to a bottom clause where the clause length is bounded by the length of the initial bottom clause. ARMGs, therefore do not need the determinacy restrictions used in Golem. ARMGs relative to a bottom clause (i.e. $armg_\perp$) have been implemented in ProGolem which combines bottom-clause construction in Progol with a Golem control strategy which uses $armg_\perp$ in place of determinate RLGG. Algorithms based on $armg_\perp$ are described and evaluated in Chapters 7 and 8.

5. **Alternative forms of bounded subsumption to address Progol's incompleteness**

   We have also studied alternative forms of subsumption relative to $\perp$ to address Progol's incompleteness. We have shown that using different conditions on the selection function which maps the literals from a clause to the literals of $\perp$, we can get different forms of subsumption orders relative to $\perp$. We have defined the refinement operator ρ_2 based on an alternative form of bounded subsumption and demonstrated how ρ_2 can address the first type of Progol's incompleteness (i.e. incompleteness due to the ordering of the literals). This has also been empirically evaluated in Chapter 8.

6. **Stochastic refinement**

   We have studied how the refinement theory and relevant concepts such as refinement operators can be adapted for a stochastic ILP search. To address this question we introduced the concept of stochastic refinement operators and adapted a framework, called stochastic refinement search. Stochastic refinement is introduced as a probability distribution over a set of clauses and can also be viewed as a Bayesian prior over the hypotheses in a stochastic ILP search. We also defined the concept of stochastic refinement search. In general a stochastic refinement search can be viewed as a Markov chain in which the next state of the search only depends on the current state. We discussed the properties of a stochastic refinement search as two well known Markovian approaches: 1) Gibbs sampling algorithm and 2) random heuristic search. As a Gibbs sampling algorithm, a stochastic refinement search iteratively generates random samples from the hypothesis space according to a posterior distribution. We have shown how a proper sample size can be selected to guarantee that with a high probability a consistent and compressive hypothesis is generated by a stochastic refinement search (with unary or binary refinement operators). We have defined a special case of random heuristic search [Vos99] called monotonic random heuristic search and showed that a stochastic refinement search can be viewed as a monotonic random heuristic search. The advantage of studying stochastic refinement search as a random heuristic search is that we can use the theoretical results from random heuristic search in order to analyse the behaviour and convergence of the search. We also discussed genetic search for learning first-order clauses and describe a framework for genetic and stochastic refinement search for bounded subsumption. We study a special case of stochastic refinement search where stochastic refinement operators and a Genetic Algorithm (GA) search are defined with respect to bounded subsumption.

7. **Algorithms and implementations of stochastic refinement and a genetic algorithm approach for the bounded subsumption**

   We described algorithms and implementations of stochastic refinement and a genetic algorithm approach for searching the hypothesis space bounded by a bottom clause (bounded subsumption). In particular we described a genetic algorithm approach which uses the encoding and operators for the bounded subsumption and can also be characterised as a stochastic refinement search. We introduced a novel binary representation for clauses in the bounded subsumption lattice. This binary representation is based on the encoding tuples $\langle K, \theta \rangle$ and the morphism with the bounded subsumption lattice described above. We discussed the properties of the binary representation and showed that lgg(least general generalisation) and mgs(most general specialisation) for the bounded subsumption lattice, i.e. $lgg_{\perp}$ and $mgs_{\perp}$, can be implemented by simple bitwise operations on the binary encoding of clauses. This representation and encoding together with genetic and stochastic refinement operators are implemented in GA-Progol, an extension of Progol in which a stochastic refinement setting can be selected instead of Progol's standard refinement setting which uses a A-like search. In addition to the genetic algorithm, we also described other stochastic refinement searches including Quick Generalisation (QG) algorithm and QG/GA search which are implemented in GA-Progol and algorithms based on Asymmetric Relative Minimal Generalisation (ARMG) which are implemented in ProGolem. QG algorithm is a stochastic algorithm which constructs maximally general consistent clauses by randomly pruning Progol bottom clauses. The QG sampling algorithm can be viewed as a stochastic refinement search with unary refinement operator which randomly samples from "fringe" clauses (i.e. maximally general consistent clauses in the hypothesis space). We describe a sampling algorithm based on QG which returns the clause with highest positive compression from a sample of s calls to QG. We also described a combination of QG and GA (i.e. QG/GA algorithm) in which the initial population of the GA consists of clauses generated by the QG algorithm. ProGolem implements an efficient (and non-determinate) variant of Golem's RLGG for the subsumption relative to $\perp$. This is called Asymmetric Relative Minimal Generalisation (ARMG) or $armg_\perp$ which similar to $lgg_\perp$ is defined based on subsumption relative to $\perp$ where the clause length is bounded by the length of the initial bottom clause. ProGolem, therefore does not need the determinacy restrictions used in Golem. ProGolem combines bottom-clause construction in Progol with a Golem control strategy which uses $armg_\perp$ in place of determinate RLGG. ProGolem uses random sampling and a beam search to construct ARMGs with new examples at each iteration. We also describe a stochastic GA-like search implemented in ProGolem. Clause refinement in ProGolem can be viewed as stochastic refinement search relative to $\perp$.

8. **Empirical evaluation**

Algorithms described above which are implemented based on the framework of stochastic refinement relative to bounded subsumption are evaluated on artificial and real-world problems. These results can be summarised as follows. We showed that using $lgg_\perp$ crossover operator can improve the convergence of the genetic search. It was shown that in the random trains problem, the genetic search can lead to significantly increased efficiency for learning complex target concepts and this can be achieved without substantial decrease of accuracy. We compared GA and A^* on a set of problems involving long target clauses with different sizes, i.e. a subset of Phase Transition (PT) dataset. The results clearly suggested the advantage of GA for learning long clauses, i.e. GA found a solution with a similar or better accuracy than A^* in significantly less time. We demonstrated that GA-Progol can find the correct solution for some special cases where the solution cannot be found by Progol's refinement operator due to its incompleteness. The experimental results comparing A^*, QG and QG/GA suggested the efficiency and accuracy advantages of QG/GA in particular in problems with long target clauses and when the density of consistent clauses ($Cs\%$) is low. The results also suggested a potential advantage of the GA-like search in some problems, i.e. the accuracy of $ARMG$ with the GA-like search on Pyrimidines dataset is significantly higher than Aleph, QG and $ARMG$ with beam search.

9.2 Conclusions

The main research questions raised in Chapter 1 can be summarised as follows:

1. What are the properties of the hypothesis space bounded by a bottom clause?
2. How can the generality order of clauses and the relevant concepts such as refinement be adapted to be used in a stochastic search?
3. Can a stochastic refinement search, e.g. in the form of a Genetic Algorithm, achieve performance improvements over existing search methods for the bounded hypothesis space, e.g. A* search in Progol?

Note that these questions are related, i.e. in order to address the third question we need to address the first and the second questions.

In order to address the first question, the concept of bounded subsumption was introduced in this thesis and it was demonstrated that Progol's refinement can be characterised with respect to this order. We also studied the lattice and cover structure of bounded subsumption and it was shown that this lattice is isomorphic to an atomic lattice and a lattice of partitions. We have also shown that, unlike for θ-subsumption, ideal refinement operators and efficient least and minimal generalisation operators (i.e. $lgg_\perp$ and $armg_\perp$) can be defined for bounded subsumption.

In order to address the second question, we introduced the concept of stochastic refinement operators and adapted a framework, called stochastic refinement search. We discussed the properties of a stochastic refinement search as two well known Markovian approaches: 1) Gibbs sampling algorithm and 2) random heuristic search. The refinement graph theory has been viewed as the main theoretical foundation of ILP [NCdW97]. Since the publication of this theory, there have been attempts to build ILP systems based on stochastic and randomised methods (e.g. [PKK93, Sri00, TNM02, RK03, ZSP06, PŽZ$^+$07, MTN07, DPZ08]). However, to date there is very little theory to support the developments of these systems. We believe the research on stochastic refinement presented in this thesis is a step in this direction.

In order to address the third question, we described and evaluated algorithms and implementations which are based on (i) the encoding and refinement operators for bounded subsumption and (ii) stochastic refinement relative to a bottom clause. We described and evaluated three different set of implementations: i) a stochastic refinement search based on Genetic Algorithms (GA), ii) Quick Generalisation (QG) and a combination of QG and GA (QG/GA) and iii) stochastic search based on Asymmetric Relative Minimal Generalisation (ARMG). The experimental results suggested that these algorithms can outperform existing search methods for the bounded hypothesis space, e.g. A* search in Progol. The results indicated the efficiency and accuracy advantages of the stochastic algorithms, in particular in problems with long target clauses. The results also suggested that a combination of these algorithms, i.e. QG/GA can provide increased accuracy over each individual approach.

Hence, we believe that the research questions raised in Chapter 1 have been addressed in this thesis. Parts of this thesis have already been published as conference and journal papers, e.g. Proceedings of the 10th, 12th, 18th, 19th and 20th International Conference on Inductive Logic Programming [TNM00], [TNM02], [TNM08], [MSTN10a] and [TNM11], Proceedings of the Genetic and Evolutionary Computation Conference

(GECCO) [TNM01] and Machine Learning Journal [MTN07] and [TNM09].

The theoretical analysis and algorithms described in this thesis can be adapted for any ILP system which uses a bottom clause or a template for generating the hypotheses. These include ILP systems which use some form of Inverse Entailment (e.g. [IY98], [Ino01], [RBR03] and [YII13]). The following are examples of related works by other authors who have used and cited parts of the research presented in this thesis (via our previously published papers listed above): learning symbolic and numeric constraints in a pattern (i.e. bottom clause) using a GA [BV01], a simulated annealing framework for ILP [SPR04b], learning theories using Estimation of Distribution Algorithms (EDA) and reduced bottom clauses [PZ12], heuristic inverse subsumption in full-clausal theories using mode-directed bottom generalisation [YII13] and fast relational learning using bottom clause professionalisation with artificial neural networks [FZG14].

Moreover, the results of this thesis have also contributed to some related research outside the scope of the thesis. These include a probabilistic ILP setting for bounded subsumption called Hypothesis Frequency Estimation (HFE) [TNBRM12] and a Bayesian meta-interpretative learning using higher-order stochastic refinement [MLCTN13]. These are discussed in the following section.

The GA approach for bounded subsumption discussed in this thesis can be compared with the GA-based systems such as GA-SMART [GS92] which require a language template extracted (manually) from background knowledge. However, as discussed in Section 6.3 there are problems for defining templates and also these systems use a very limited form of background knowledge. In this thesis we described a genetic algorithm approach in which a template (i.e. a bottom clause) is automatically constructed using ILP methods (e.g. Inverse Entailment). Hence, in the hybrid GA-ILP framework described in this thesis, background knowledge is used in the same way as in the ILP systems such as Progol and Aleph. This GA-ILP framework not only uses a standard ILP representation (unlike the GA-based systems discussed in Section 6.3) but we also showed that the proposed encoding is isomorphic to the bounded subsumption lattice. This property can be used to design task-specific genetic operators.

There are however several limitations about the proposed framework, algorithms and implementations discussed in this thesis. We discuss these limitations together with some work in progress and further research in the next section.

9.3 Further research

In this section we discuss some work in progress and further research related to this thesis. These are mainly related to addressing some limitations of the proposed framework, algorithms and implementations discussed in this thesis. Some of these limitations are inherited from the ILP setting which we have considered in this thesis (i.e. Progol's ILP setting) and some are related to our specific implementations.

The first limitation which we discuss in this section is related to the single clause assumption which has been inherited from Progol's ILP setting. The next limitation is related to the sensitivity of Progol-like algorithms to the order of examples. This is related to the fact that bottom-clauses which define the hypothesis space (bounded subsumption) are generated from the next seed example and in some problems this could affect the solution found by these systems. Again, this is inherited from Progol's ILP setting. The other limitation is related to the specific implementation of the partition-based encoding proposed in this thesis. In the following we describe these limitations in more details and also discuss work in progress and further research to address each limitation.

Learning clausal theories and Meta-Interpretive Learning (MIL). The theoretical analyses and implementations described in this thesis use a single clause assumption which has been inherited from Progol's ILP setting. For example each node in the bounded subsumption lattice is a single clause and each individual in the GA population stands for a single clause. The final hypothesis consists of clauses each corresponding to an iteration of the cover-set algorithm. There are alternative approaches (e.g. [PŽZ$^+$07], [MLTN12] and [YII13]) to Progol's greedy cover-set algorithm which consider clausal theories rather single clauses at a time. However, the refinement and search space for clausal theories are more complex than that for single clauses. The approach which we have taken to extend the results of this thesis to clausal theories is based on a new ILP setting called Meta-Interpretive Learning (MIL) [MLPTN14]. In the MIL setting clausal theories are represented as a single higher-order clause. Hence, learning first-order clausal theories can be achieved by learning single higher-order clauses in the MIL setting. In other words, the MIL setting can be viewed as a bridge to extend the theories on single clause refinement (and stochastic refinement) to the clausal theories. It has been shown (Propositions 2 and 4 in [MLPTN14]) that single higher-order clauses in MIL form a subsumption lattice which is bounded by a single higher-order clause. The initial MIL setting which was only applied to grammatical inference has been extended to dyadic definite clauses [ML13]. The stochastic refinement described in this thesis has been already adapted [MLCTN13] for the MIL setting. As future work we would like extend other results from this thesis to the MIL setting, in particular the analysis of bounded (higher-order) subsumption lattice and a genetic algorithms approach to learning clausal theories. From a GA point of view, this setting would be similar to the Pittsburgh approach [Smi83, JSG93] in which each individual in the population encodes a set of rules.

A probabilistic learning and inference approach for bounded subsumption.

In this thesis we studied the bounded subsumption and stochastic search algorithms in a Progol-like ILP setting. In this setting a bottom clause is constructed from the next positive example and the subsumption lattice bounded by this bottom clause defines the search space. Because of Progol's set-covering algorithm, the output of the learning is sometime sensitive to the initial order of the examples. The approach described here was initially developed to address this problem by averaging over different permutations of the examples, however, it turned out to be useful as a simple probabilistic ILP approach which could also be used in other similar ILP settings.

This method can be used whenever training examples act as seeds to define the hypothesis space, e.g. in the ILP systems where a most specific clause is built from the next positive example. Hence, different permutations of the training examples define different parts of the hypothesis space. We used this property to sample from the hypothesis space by random permutations of the training data. Probability of hypotheses can be estimated based on the frequency of occurrence when random permutations of the training data (and hence different seeds for defining the hypothesis space) are considered. We have developed a probabilistic ILP technique called Hypothesis Frequency Estimation (HFE) [TNBRM12] based on the method described above.

HFE has been used in [BCLM$^+$11] and [TNMR$^+$13] to learn probabilistic food-webs from ecological data. In this problem abductive ILP has been used to learn ground hypotheses in the form of *eats* relations between different species from training data on the abundance of different species following an agricultural management. The set of ground hypotheses can be visualised as a network of trophic links (food webs). In this network a ground fact $eats(a, b)$ is represented by a trophic link from species b to species a. Using HFE, the probabilities or the thickness of trophic links are estimated based on the frequency of occurrence from random permutations of the training data. This probabilistic network can also be represented using standard probabilistic representations in ILP such as SLPs [Mug96] or ProbLog [DKT07]. For this we can use relative frequencies in the same way probabilities are used in probabilistic ILP. We can then use the probabilistic inferences based on these representations to estimate other probabilities. We have used this method in the leave-one-out experiments in [TNBRM12] to evaluate probabilistic tropic networks and compare them with non-probabilistic networks. The results showed that the predictive accuracies for the non-probabilistic networks are significantly lower than the probabilistic networks. This suggests that using the permutation based HFE method for estimating probabilities of hypotheses leads to significant increased predictive accuracies compared to the non-probabilistic hypotheses.

HFE can be viewed as a method for sampling hypotheses and prediction based on model averaging for the hypothesis space bounded by a bottom clause. This method is based on direct sampling from the hypothesis space and can be used to build an average model from possible worlds. This can also be viewed as an approximations to Bayes optimal classifier and can be used for prediction.

We have only evaluated HFE approach for ground hypotheses in an abductive setting. However, the same approach can be used to estimate the probabilities for non-ground hypotheses based on sampling from the bounded hypothesis space. As future work we would like to apply HFE in applications involving non-ground hypotheses and also to integrate HFE into the stochastic refinement and search algorithms described in this thesis for bounded subsumption.

GA representation: binary versus partition-based encoding. The binary representation used in GA-Progol is one possible implementation of the partition-based encoding described in Chapter 5. As mentioned in Chapter 7, this binary representation is redundant, i.e. there is a many-to-one genotype-to-phenotype mapping. A redundant representation is not regarded as a serious problem in GAs and some authors (e.g. [Alt95] and [RG03]) suggest that under some conditions a redundant representation can even improve the genetic search. Nevertheless, the redundancy in our proposed binary representation of variable partitions can be avoided using a different encoding. GA-Progol also includes an implementation based on Grouping Genetic Algorithms (GGA) [Fal98]. However, this has not been thoroughly tested and as future work we intend to complete and evaluate this implementation.

Fast subsumption testing for bounded subsumption. In this thesis we have shown the morphism between the bounded subsumption lattice and an atomic lattice. As mentioned in Chapter 4 an atomic subsumption testing can be reduced to a unification problem which can be decided in linear time [GL85]. This means that efficient subsumption testing algorithms might be designed for clauses in bounded subsumption. This subject was not investigated in this thesis and could be a topic for future research.

Bibliography

[AD08] A. Anglade and S. Dixon. Characterisation of harmony with inductive logic programming. In *International Conference on Music Information Retrieval (ISMIR)*, pages 63–68, 2008.

[AGLS98] C. Anglano, A. Giordana, G. Lo Bello, and L. Saitta. An experimental evaluation of coevolutive concept learning. In J. Shavlik, editor, *Proceedings of the 15th International Conference on Machine Learning*, pages 19–27. Morgan Kaufmann, 1998.

[AK04] M. Arias and R. Khardon. Bottom-up ILP using large refinement steps. *Proceedings of the International Conference on Inductive Logic Programming*, pages 26–43, 2004.

[Ala08] J. Alander. Indexed bibliography of genetic algorithms papers. Technical Report 94-1-FTP, Department of Information Technology and Production Economics, University of Vaasa, 2008.

[Alt95] L. Altenberg. Genome growth and the evolution of the genotype-phenotype map. In *Evolution and biocomputation*, pages 205–259. Springer, 1995.

[Ant89] J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In *Proc. of 3rd International Conference on Genetic Algorithms*, pages 86–91. Morgan Kaufmann, 1989.

[AR99] E. Alphonse and C. Rouveirol. Object identity for relational learning. Technical report, Deliverable LRIc (1), ESPRIT LTR 20237 (ILP2), 1999.

[ARUK12] D. Alrajeh, A. Russo, S. Uchitel, and J. Kramer. Integrating model checking and inductive logic programming. In *Proceedings of the 21st International Conference on Inductive Logic Programming*, LNAI 7207, pages 45–60, 2012.

[ASHMS07] A. Amini, H. Lodhi S. H. Muggleton, and M. J. E. Sternberg. A novel logic-based approach for quantitative toxicology prediction. *Journal of Chemical Information and Modelling*, 47(3):998–1006, 2007. DOI: 10.1021/ci600223d.

[Bac20] F. Bacon. Novum organum. *Open Court, Chicago, IL, 1994. Edited and translated by P. Urbach and J. Gibson*, First published in 1620.

[Bäc95] T. Bäck. *Evolutionary Algorithms in theory and practice.* New-York:Oxford University Press, 1995.

[Bar75] J. Barnes. *Aristotle's Posterior Analytics.* Oxford University Press, 1975.

[BCLM$^+$11] D. A. Bohen, G. Caron-Lormier, S. H. Muggleton, A. Raybould, and A. Tamaddoni-Nezhad. Automated discovery of food webs from ecological data using logic-based machine learning. *PloS ONE*, 6(12), 2011.

[BD97] H. Blockeel and L. De Raedt. Lookahead and discretisation in ILP. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 77–84. Springer-Verlag, Berlin, 1997. LNAI 1297.

[BG95] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1995.

[BGD02] L. Bianchi, L. M. Gambardella, and M. Dorigo. An ant colony optimization approach to the probabilistic traveling salesman problem. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 883–892, London, UK, 2002. Springer-Verlag.

[BGSS03] M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational learning as search in a critical region. *J. Mach. Learn. Res.*, 4:431–463, 2003.

[BMV91] I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In *Proceedings of the Eighth International Machine Learning Workshop*, San Mateo, Ca, 1991. Morgan-Kaufmann.

[BP00] S. Boettcher and A. G. Percus. Solving constraint satisfaction problems with heuristic-based evolutionary algorithms. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 1578–1584, California, USA, 2000. IEEE Press.

[BS99] L. Badea and M. Stanciu. Refinement operators can be (weakly) perfect. *Proceedings of the 9th International Workshop on Inductive Logic Programming*, 1634:21–32, 1999.

[BV01] A. Braud and C. Vrain. A genetic algorithm for propositionalization. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 27–40. Springer, 2001.

[BV09] Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347–364, 2009.

[BZB01] R. Basilio, G. Zaverucha, and V. C. Barbosa. Learning logic programs with neural networks. *Proceedings of the International Conference on Inductive Logic Programming*, pages 15–26, 2001.

[CD97] W. Cohen and P. Devanbu. A Comparative Study of ILP Methods for Software Fault Prediction. In *Proceedings of the 14th International Conference on Machine Learning*, pages 66–74. Morgan Kaufmann, 1997.

[CHH$^+$02] J. Cheng, C. Hatzis, H. Hayashi, M-A. Krogel, S. Morishita, D. Page, and J. Sese. Kdd cup 2001 report. *SIGKDD Explorations*, 3(2):47–64, 2002.

[CKP$^+$06] A. Cocora, K. Kersting, C. Plagemann, W. Burgard, and L. De Raedt. Learning relational navigation policies. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2792–2797. IEEE, 2006.

[CMS03] A. Cootes, S. H. Muggleton, and M. J. E. Sternberg. The automatic discovery of structural principles describing protein fold space. *Journal of Molecular Biology*, 2003.

[CP00] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Cus97] J. Cussens. Part-of-speech tagging using Progol. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 93–108. Springer-Verlag, 1997.

[DB91] L. De Raedt and M. Bruynooghe. CLINT: A multistrategy interactive concept-learner and theory revision system. In *Proceedings of the 1st International Workshop on Multistrategy Learning*, pages 175–191. Morgan Kaufmann, 1991.

[DCM00] S. Dzeroski, J. Cussens, and S. Manandhar. An Introduction to Inductive Logic Programming and Learning Language in Logic. In James Cussens and Saso Dzeroski, editors, *Learning Language in Logic*, volume 1925 of *Lecture Notes in Computer Science*, pages 3–35. Springer-Verlag, June 2000.

[DD97] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99–146, 1997.

[DDRW94] S. Džeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of river water quality data using machine learning. In P. Zannetti, editor, *Proceedings of the 5th International Conference on the Development and Application of Computer Techniques to Environmental Studies, Vol. I: Pollution modelling*, pages 129–137, 1994.

[De 96] L. De Raedt. *Advances in Inductive Logic Programming.* IOS Press, 1996.

[De 08] L. De Raedt. *Logical and Relational Learning.* Springer-Verlag, 2008.

[DKP$^+$06] P. S. Domingos, S. Kok, H. Poon, M. Richardson, and P. Singla. Unifying logical and statistical AI. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence, AAAI06*, pages 2–7. AAAI Press / The MIT Press, 2006.

[DKT07] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A Probabilistic Prolog and its Applications in Link Discovery. In R. Lopez de Mantaras and M.M Veloso, editors, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 804–809, 2007.

[DM92] B. Dolšak and S. Muggleton. The application of inductive logic programming to finite-element mesh design. In S. Muggleton, editor, *Inductive Logic Programming*, pages 453–472. Academic Press, 1992.

[Dov95] M. Dovey. Analysis of Rachmaninoff's piano performances using Inductive Logic Programming. In *Proceedings of the 8th European Conference on Machine Learning*, pages 279–282. Springer, 1995.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[DPZ08] A. L. Duboc, A. Paes, and G. Zaverucha. Using the Bottom Clause and Mode Declarations on FOL Theory Revision from Examples. In *Proceedings of the 18th international conference on Inductive Logic Programming*, pages 91–106. Springer, 2008.

[DT99] L. Dehaspe and H. Toivonen. Discovery of frequent Datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[ELMS96] F. Esposito, A. Laterza, D. Malerba, and G. Semeraro. Refinement of Datalog programs. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming*, pages 73–94. Citeseer, 1996.

[Fal98] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, Inc., 1998.

[Fen92] C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[FMPS98] P. Finn, S. H. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the Inductive Logic Programming system Progol. *Machine Learning*, 30:241–271, 1998.

[FPS01] G. Folino, C. Pizzuti, and G. Spezzano. Parallel hybrid method for SAT that couples genetic algorithms and local search. *IEEE-EC*, 5:323–334, 2001.

[FZG14] M. V.M. França, G. Zaverucha, and A. S. Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94:81–104, 2014.

[GCSR03] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2003.

[Ghe94] K. Ghedira. Dynamic partial constraint satisfaction by multi-agent and simulated annealing. In T. Schiex and C. Bessiére, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, 1994.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GL85] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *J. ACM*, 32(2):280–295, 1985.

[GMS01] M. Guntsch, M. Middendorf, and H. Schmeck. An ant colony optimization approach to dynamic TSP. In L. Spector, E.D. Goodman, A. Wu, W. B. Langdon, H Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 860–867, San Francisco, California, USA, 2001. Morgan Kaufmann.

[GN96] A. Giordana and F. Neri. Search-intensive concept induction. *Evolutionary Computation Journal*, 3(4):375–416, 1996.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, MA, 1989.

[Gol02] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[GS92] A. Giordana and C. Sale. Learning Structured Concepts Using Genetic Algorithms. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Workshop on Machine Learning*, pages 169–178. Morgan Kaufmann, 1992.

[GS00] A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.

[GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1-2):67–100, 2000.

[Hek98] J. Hekanaho. DOGMA: A GA-Based Relational Learner. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, pages 205–214. Springer-Verlag, 1998.

[HHW96] T. Hogg, B. A. Huberman, and C. P. Williams. Phase transitions and the search problem. *Artif. Intell.*, 81(1-2):1–15, 1996.

[HKS94] D. Haussler, M Kearns, and R. Shapire. Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. *Machine Learning Journal*, 14(1):83–113, 1994.

[Hol75] J. H. Holland. *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[IFN10] K. Inoue, K. Furukawa, and I. Kobayashiand H. Nabeshima. Discovering rules by meta-level abduction. In L. De Raedt, editor, *Proceedings of the Nineteenth International Conference on Inductive Logic Programming (ILP09)*, pages 49–64, Berlin, 2010. Springer-Verlag. LNAI 5989.

[Ino01] K. Inoue. Induction, abduction and consequence-finding. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh International Workshop on Inductive Logic Programming (ILP01)*, pages 65–79, Berlin, 2001. Springer-Verlag. LNAI 2157.

[IY98] K. Ito and A. Yamamoto. Finding hypotheses from examples by computing the least generlisation of bottom clauses. In S. Arikawa and H. Motoda, editors, *Proceedings of Discovery Science '98*, pages 303–314. Springer, Berlin, 1998. LNAI 1532.

[Jan93] C. Z. Janikow. A Knowledge-Intensive Genetic Algorithm for Supervised Learning. *Machine Learning*, 13:189–228, 1993.

[JS89] K. A. De Jong and W. M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 124–132. Morgan Kaufmann, 1989.

[JSG93] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using Genetic Algorithms for Concept Learning. *Machine Learning*, 13:161–188, 1993.

[Kar95] A. Karalič. First-order regression: Applications in real-world domains. In *Proceedings of the 2nd International Workshop on Artificial Intelligence Techniques*, 1995.

[KD01] K. Kersting and L. De Raedt. Towards Combining Inductive Logic Programming with Bayesian Networks. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, LNAI 2157, pages 118–131, Berlin, 2001. Springer-Verlag.

[KDK97] B. Kompare, S. Džeroski, and A. Karalič. Identification of the Lake of Bled ecosystem with the artificial intelligence tools M5 and FORS. In *Proc. Fourth International Conference on Water Pollution*, pages 789–798. Computational Mechanics Publications, Southampton, 1997.

[KGC99] C. J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 1–6. Springer Verlag, April 1999.

[Kli94] V. Klingspor. GRDT: Enhancing model-based learning for its application in robot navigation. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 107–122. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[KM90] Y. Kodratoff and R. Michalski. Research in machine learning: Recent progress, classification of methods and future directions. In Y. Kodratoff and R. Michalski, editors, *Machine learning: an Artificial Intelligence approach*, volume 3, pages 3–30. Morgan Kaufman, San Mateo, CA, 1990.

[KMLS92] R.D. King, S.H. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of Inductive Logic Programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89(23):11322–11326, 1992.

[KMSS96] R. D. King, S. H. Muggleton, A. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives to predict mutagenicity by Inductive Logic Programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.

[KOHH06] M. Kuwabara, T. Ogawa, K. Hirata, and M. Harao. On generalization and subsumption for ordered clauses. In *Proceedings of the JSAI 2005 Workshops*, pages 212–223, 2006.

[Kow80] R. A. Kowalski. *Logic for Problem Solving.* North Holland, 1980.

[Koz91] J. R. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1991.

[KS90] R. King and M. Sternberg. Machine learning approach for the prediction of protein secondary structure. *J. Mol. Biol.*, (216):441–457, 1990.

[KSS95] R. D. King, A. Srinivasan, and M. J. E. Sternberg. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13:411–433, 1995.

[KWJ$^+$04] R. D. King, K. E. Whelan, F. M. Jones, P. K. G. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427:247–252, 2004.

[Lai87] P. D. Laird. *Learning from good data and bad.* PhD thesis, Yale University, 1987.

[LD93] N. Lavrač and S. Džeroski. *Inductive Logic Programming : Techniques and Applications.* Ellis Horwood, 1993.

[LD04] S. D. Lee and L. De Raedt. Constraint Based Mining of First Order Sequences in SeqLog. *Database Support for Data Mining Applications*, pages 154–173, 2004.

[LDG91] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning non-recursive definitions of relations with LINUS. In Yves Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning, volume 482 of Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1991.

[Lee06] S. D. Lee. *Constrained Mining of Patterns in Large Databases.* PhD thesis, Albert-Ludwigs-University, 2006.

[Llo95] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[LM10] H. M. Lodhi and S. H. Muggleton, editors. *Elements of Computational Systems Biology*. Wiley, New Jersey, 2010.

[LMS10] H. Lodhi, S. H. Muggleton, and M. J. E. Sternberg. Multi-class protein fold recognition using large margin logic based divide and conquer learning. *SIGKDD Exploration*, 11(2):117–122, 2010.

[LW95] K. S. Leung and M. L. Wong. Genetic Logic Programming and Applications. *IEEE Expert*, 10(5):68–76, 1995.

[MB88] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.

[MB00] S.H. Muggleton and C.H. Bryant. Theory completion using inverse entailment. In *Proc. of the 10th International Workshop on Inductive Logic Programming (ILP-00)*, pages 130–146, Berlin, 2000. Springer-Verlag.

[McC59] J. McCarthy. Programs with commonsense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91. London, 1959.

[MD94] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[MF90] S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, Tokyo, 1990. Ohmsha.

[Mic80] R. S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Trans. on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.

[Mit82] T. M. Mitchell. Generalisation as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mit97] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[Mit98] M. Mitchell. *An introduction to Genetic Algorithms*. MIT Press, 1998.

[MKKC86] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[MKS92] S. H. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[ML13] S. H. Muggleton and D. Lin. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. In *Proceedings of the 23rd International Joint Conference Artificial Intelligence (IJCAI 2013)*, pages 1551–1557, 2013.

[MLCTN13] S. H. Muggleton, D. Lin, J. Chen, and A. Tamaddoni-Nezhad. MetaBayes: Bayesian Meta-Interpretative Learning using Higher-Order Stochastic Refinement. In *Proceedings of the 23rd International Conference on Inductive Logic Programming*, 2013. accepted.

[MLPTN14] S. H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94:25–49, 2014.

[MLTN12] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. MC-Toplog: Complete multi-clause learning guided by a top theory. In *Proceedings of the 21st International Conference on Inductive Logic Programming*, LNAI 7207, pages 238–254, 2012.

[MM99] S. Muggleton and F. Marginean. Binary refinement. In *Proceedings of Workshop on Logic-Based Artificial Intelligence*, 1999.

[MMHL86] R. Michalski, I. Mozetic, J. Hong, and N. Lavrač. The AQ15 inductive learning system: an overview and experiments. In *Proceedings of IMAL 1986*, Orsay, 1986. Université de Paris-Sud.

[Mon93] D. J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993.

[Moo97] R. J. Mooney. Inductive logic programming for natural language processing. In S. Muggleton, editor, *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, pages 3–21. Springer-Verlag, Berlin, 1997. LNAI 1314.

[Moy02] S. Moyle. Using theory completion to learn a robot navigation control program. In *Proceedings of the 12th International Conference on Inductive Logic Programming*, pages 182–197. Springer-Verlag, 2002.

[MRP$^+$11] S. H. Muggleton, L. De Rate, D. Poole, I. Bratko, P. Flach, and K. Inoue. ILP turns 20: biography and future challenges. *Machine Learning*, 2011. Published online, DOI: 10.1007/s10994-011-5259-2.

[MSTN10a] S. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. ProGolem: a system based on relative minimal generalisation. In *Proceedings of the 19th International Conference on Inductive Logic Programming (ILP09)*, pages 131–148. Springer, 2010.

[MSTN10b] S. H. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. TopLog: ILP using a logic program declarative bias. In *Proceedings of the International Conference on Logic Programming 2008*, LNCS 5366, pages 687–692. Springer-Verlag, 2010.

[MTN07] S. H. Muggleton and A. Tamaddoni-Nezhad. QG/GA: A stochastic search for Progol. *Machine Learning*, 70(2–3):123–133, 2007.

[MTNW03] S. H. Muggleton, A. Tamaddoni-Nezhad, and H. Watanabe. Induction of enzyme classes from biological databases. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, pages 269–280. Springer-Verlag, 2003.

[Mug87] S. H. Muggleton. Duce, an oracle based approach to constructive induction. In *IJCAI-87*, pages 287–292. Kaufmann, 1987.

[Mug91] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[Mug92] S. H. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.

[Mug94] S. Muggleton. Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, 1994.

[Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[Mug96] S. H. Muggleton. Stochastic Logic Programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

[NCdW97] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming.* Springer-Verlag, Berlin, 1997. LNAI 1228.

[Pei32] C. S. Peirce. Elements of logic. In C. Hartshorne and P. Weiss, editors, *Collected Papers of Charles Sanders Peirce*, volume 2. Harvard University Press, Cambridge, MA, 1932.

[PKK93] U. Pompe, M. Kovacic, and I. Kononenko. SFOIL: Stochastic approach to inductive logic programming. In *Proceedings of the Second Electrotechnical and Computer Science Conference ERK*, volume 93, pages 189–192. Citeseer, 1993.

[Plo69] G. D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.

[Plo71] G. D. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, August 1971.

[PS03] D. Page and A. Srinivasan. ILP: a short look back and a longer look forward. *J. Mach. Learn. Res.*, 4:415–430, 2003.

[PW02] M. L. Pilat and T. White. Using genetic algorithms to optimize ACS-TSP. *Lecture Notes in Computer Science*, 2463:282–287, 2002.

[PZ12] C. G. Pitangui and G. Zaverucha. Learning theories using estimation distribution algorithms and (reduced) bottom clauses. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 286–301. Springer, 2012.

[PŽZ+07] A. Paes, F. Železný, G. Zaverucha, D. Page, and A. Srinivasan. ILP through propositionalization and stochastic k-term DNF learning. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 379–393. Springer, 2007.

[QR89] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the Minimum Description Length principle. *Information and Computation*, 80:227–248, 1989.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Rad91] N. J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–20, 1991.

[RBR03] O. Ray, K. Broda, and A. Russo. Hybrid Abductive Inductive Learning: a Generalisation of Progol. In *13th International Conference on Inductive Logic Programming*, volume 2835 of *LNAI*, pages 311–328. Springer Verlag, 2003.

[Rey69] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, Edinburgh, 1969.

[RG03] F. Rothlauf and D. E. Goldberg. Redundant representations in evolutionary computation. *Evolutionary Computation*, 11(4):381–415, 2003.

[Ris78] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[RK03] U. Ruckert and S. Kramer. Stochastic Local Search in k-term DNF Learning. In *Proc. 20th International Conf. on Machine Learning*, pages 648–655, 2003.

[RKD02] U. Ruckert, S. Kramer, and L. De Raedt. Phase Transitions and Stochastic Local Search in k-Term DNF Learning. In H. Toivonen T. Elomaa, H. Mannila, editor, *Machine Learning: ECML 2002*, pages 405–417. Springer, 2002.

[RN10] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Englewood Cliffs, NJ, 2010.

[RW00] A. M. Richard and C. R. Williams. Distributed structure-searchable toxicity (DSSTox) public database network: A proposal. *Mutation Research*, 499:27–52, 2000.

[SAK95] G. Venturini S. Augier and Y. Kodratoff. Learning first order logic rules with a genetic algorithm. In *Proceedings of the 1st International Conference on Knowledge Discovery in DataBases*, pages 21–26. AAAI, 1995.

[Sam99] G. Sampath. A modified simulated annealing algorithm and its application to the satisfiability problem. *CSSTAT: Computing Science and Statistics, Interface Foundation of North America*, 31, 1999.

[San10] J. C. A. Santos. *Efficient Learning and Evaluation of Complex Concepts in Inductive Logic Programming.* PhD thesis, Dept. of Computing, Imperial College London, 2010.

[Sat05] T. Sato. Generative modeling with failure in prism. In *International Joint Conference on Artificial Intelligence*, pages 847–852. Morgan Kaufmann, 2005.

[SB81] HP Sankappanavar and S. Burris. *A course in universal algebra.* Springer-Verlag, 1981.

[SB86] C. Sammut and R. B. Banerji. Learning concepts by asking questions. In R. Michalski, J. Carbonnel, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach. Vol. 2*, pages 167–192. Kaufmann, Los Altos, CA, 1986.

[SDI09] G. Synnaeve, A. Doncescu, and K. Inoue. Kinetic models for logic-based hypothesis finding in metabolic pathways. In *Proceedings of the 19th International Conference on Inductive Logic Programming*, 2009.

[SGC11] L. Saitta, A. Giordana, and A. Cornuéjols. *Phase Transitions in Machine Learning.* Cambridge University Press, 2011.

[Sha83] E. Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

[SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, pages 337–343. AAAI Press, 1994.

[SLM92] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.

[SM03] M. J. E. Sternberg and S. H. Muggleton. Structure activity relationships (SAR) and pharmacophore discovery using inductive logic programming (ILP). *QSAR and Combinatorial Science*, 22, 2003.

[Smi83] S. F. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proc. 8th Int. Joint Conf. on A.I.*, pages 422–425, 1983.

[SMKS96] A. Srinivasan, S. H. Muggleton, R. King, and M. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.

[SMS97] A. Srinivasan, , R. D. King S. H. Muggleton, and M. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 273–287. Springer-Verlag, Berlin, 1997. LNAI 1297.

[SN00] L. Schoofs and B. Naudts. Ant colonies are good at solving constraint satisfaction problems. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 1190–1195, California, USA, 2000. IEEE Press.

[SNCDP12] J. C. A. Santos, H. Nassif, and S. H. Muggleton M. J. E. Sternberg C. D. Page. Automated identification of protein-ligand interaction features using Inductive Logic Programming: A hexose binding case study. *BMC Bioinformatics*, 13(162), 2012.

[SPCK06] A. Srinivasan, D. Page, R. Camacho, and R. King. Quantitative pharmacophore models with Inductive Logic Programming. *Machine learning*, 64(1):65–90, 2006.

[SPR04a] M. Serrurier, H. Prade, and G. Richard. A simulated annealing framework for ILP. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 288–304. Springer-Verlag, 2004.

[SPR04b] M. Serrurier, H. Prade, and G. Richard. A simulated annealing framework for ilp. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 288–304. Springer, 2004.

[SR00] M. Sebag and C. Rouveirol. Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning Journal*, 38:43–65, 2000.

[Sri99] A. Srinivasan. A study of two sampling methods for analyzing large datasets with ilp. *Data Min. Knowl. Discov.*, 3(1):95–123, 1999.

[Sri00] A. Srinivasan. A study of two probabilistic methods for searching large spaces with ilp. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, Oxford, 2000.

[Sri05] A. Srinivasan. Five problems in five areas for five years. In Stefan Kramer and Bernhard Pfahringer, editors, *Proceedings of the 15th International Conference on Inductive Logic Programming*, volume 3625 of *Lecture Notes in Artificial Intelligence*, page 424. Springer-Verlag, 2005.

[Sri07] A. Srinivasan. *The Aleph Manual.* University of Oxford, 2007.

[SSJ$^+$09] L. Specia, A. Srinivasan, S. Joshi, G. Ramakrishnan, and M. das Graças Volpe Nunes. An investigation into feature construction to assist word sense disambiguation. *Machine learning*, 76(1):109–136, 2009.

[Ste93] J. Stender. *Parallel Genetic Algorithms: Theory and Practice.* IOS Press, Amsterdam, 1993.

[STNL$^+$13] M. J. E. Sternberg, A. Tamaddoni-Nezhad, V.I. Lesk, , E. Kay, P.G. Hitchen, A. Cootes, L.B. Alphen, M.P. Lamoureux, H.C. Jarrell, C.J. Rawlings, E.C. Soo, C.M. Szymanski, A. Dell, B.W. Wren, and S.H. Muggleton. Gene Function Hypotheses for the Campylobacter jejuni Glycome Generated by a Logic-Based Approach. *Journal of Moleular Biology*, 425(1):186–197, 2013.

[TMS98] M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. Protein fold recognition. In C. D. Page, editor, *Proc. of the 8th International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 53–64, Berlin, 1998. Springer-Verlag.

[TNBRM12] A. Tamaddoni-Nezhad, D. Bohan, A. Raybould, and S. H. Muggleton. Machine learning a probabilistic network of ecological interactions. In *Proceedings of the 21st International Conference on Inductive Logic Programming*, LNAI 7207, pages 332–346, 2012.

[TNCK^{+}07] A. Tamaddoni-Nezhad, R. Chaleil, A. Kakas, M. Sternberg, J. Nicholson, and S.H. Muggleton. Modeling the effects of toxins in metabolic networks. *IEEE Engineering in Medicine and Biology*, 26:37–46, 2007.

[TNCKM06] A. Tamaddoni-Nezhad, R. Chaleil, A. Kakas, and S.H. Muggleton. Application of abductive ILP to learning metabolic network inhibition from temporal data. *Machine Learning*, 64:209–230, 2006.

[TNKMP04] A. Tamaddoni-Nezhad, A. Kakas, S.H. Muggleton, and F. Pazos. Modelling inhibition in metabolic pathways through abduction and induction. In *Proceedings of the 14th International Conference on Inductive Logic Programming*, pages 305–322. Springer-Verlag, 2004.

[TNM00] A. Tamaddoni-Nezhad and S. H. Muggleton. Searching the subsumption lattice by a genetic algorithm. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 243–252. Springer-Verlag, 2000.

[TNM01] A. Tamaddoni-Nezhad and S. H. Muggleton. Using Genetic Algorithms for Learning Clauses in First-Order Logic. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, pages 639–646, San Francisco, CA, 2001. Morgan Kaufmann Publishers.

[TNM02] A. Tamaddoni-Nezhad and S. H. Muggleton. A genetic algorithms approach to ILP. In *Proceedings of the 12th International Conference on Inductive Logic Programming*, pages 285–300. Springer-Verlag, 2002.

[TNM08] A. Tamaddoni-Nezhad and S. H. Muggleton. A note on refinement operators for IE-based ILP systems. In *Proceedings of the 18th International Conference on Inductive Logic Programming*, LNAI 5194, pages 297–314. Springer-Verlag, 2008.

[TNM09] A. Tamaddoni-Nezhad and S. H. Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76(1):37–72, 2009.

[TNM11] A. Tamaddoni-Nezhad and S. Muggleton. Stochastic Refinement. In *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP10)*, pages 222–237. Springer-Verlag, 2011.

[TNMR$^+$13] A. Tamaddoni-Nezhad, G. Milani, A. Raybould, S. Muggleton, and D. Bohan. Construction and validation of food-webs using logic-based machine learning and text-mining. *Advances in Ecological Research*, 49:225–289, 2013.

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):435–460, 1950.

[Var93] A. Varšek. *Inductive Logic Programming with Genetic Algorithms.* PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia, 1993.

[VBD97] E. Van Baelen and L. De Raedt. Analysis and prediction of piano performances using Inductive Logic Programming. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 55–71. Springer, 1997.

[vdL95] P. van der Laag. *An Analysis of Refinement Operators in Inductive Logic Programming.* Tinbergen Institute Research Series, Rotterdam, 1995.

[vdLNC94] P. R. J. van der Laag and S-H. Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In *Machine Learning: ECML-94: European Conference on Machine Learning, Catania, Italy, April 6-8, 1994: Proceedings*, pages 307–322. Springer, 1994.

[Ver75] S.A. Vere. Induction of concepts in the predicate calculus. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 282–287. Morgan Kaufmann, 1975.

[VL91] M. D. Voset and G. E. Liepinsl. Punctuated equilibria in genetic search. *Complex systems*, 5:31–44, 1991.

[Vos91] M. D. Vose. Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence*, 50(3):385–396, 1991.

[Vos95] M. D. Vose. Modeling simple genetic algorithms. *Evolutionary Computation*, 3(4):453–472, 1995.

[Vos99] M. D. Vose. Random heuristic search. *Theoretical Computer Science*, 229(1):103–142, 1999.

[Win70] P. H. Winston. *Learning Structural Descriptions from Examples.* PhD thesis, MIT, Cambridge, Massachusetts, January 1970.

[WL97] M. L. Wong and K. S. Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, 1997.

[YII13] Y. Yamamoto, K. Inoue, and K. Iwanuma. Heuristic inverse subsumption in full-clausal theories. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 241–256. Springer, 2013.

[ZM93] J. M. Zelle and R. J. Mooney. ILP techniques for learning semantic grammars. In F. Bergadano, L. De Raedt, S. Matwin, and S. Muggleton, editors, *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, pages 83–92. Morgan Kaufmann, 1993.

[ZSP03] F. Zelezny, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 333–345. Springer-Verlag, 2003.

[ZSP06] F. Zelezny, A. Srinivasan, and D. Page. Randomised restarted search in ILP. *Machine Learning, 64*, 1(3):183–208, 2006.

[ŽŽGS$^+$07] M. Žáková, F. Železnỳ, J. Garcia-Sedano, C. Masia Tissot, N. Lavrač, P. Křemen, and J. Molina. Relational data mining applied to virtual engineering of product designs. In *Proceedings of the 16th International Conference on Inductive Logic Programming*, pages 439–453. Springer, 2007.

Appendix A

Progol's algorithms

The following are Progol's algorithms for constructing the bottom clause ($\perp_i$), cover set algorithm and A^*-like algorithm for finding a clause with maximal compression as described in [Mug95].

Algorithm A.1 Algorithm for constructing $\perp_i$.

1. Given natural numbers h, i, Horn clauses B, definite clause e and set of mode declarations M.

2. Let $k = 0$, $hash : Terms \rightarrow N$ be a hash function which uniquely maps terms to natural numbers, $\overline{e}$ be the clause normal form logic program $\overline{a} \wedge b_1 \wedge .. \wedge b_n$, $\perp_i = \langle\rangle$ and InTerms$= \emptyset$.

3. If there is no modeh in M such that $a(m) \preceq a$ then return $\Box$. Otherwise let m be the first modeh declaration in M such that $a(m) \preceq a$ with substitution θ_h. Let a_h be a copy of $a(m)$ and for each v/t in θ_h if v corresponds to a #type in m then replace v in a_h by t otherwise replace v in a_h by v_k where $k = hash(t)$ and add v to InTerms if v corresponds to +type. Add a_h to $\perp_i$.

4. If $k = i$ return $\perp_i$ else $k = k + 1$.

5. For each modeb m in M let $\{v_1, .., v_n\}$ be the variables of +type in $a(m)$ and $T(m) = T_1 \times .. \times T_n$ be a set of n-tuples of terms such that each T_i corresponds to the set of all terms of the type associated with v_i in m (term t is tested to be of a particular type by calling Prolog with type(t) as goal). For each $\langle t_1, .., t_n\rangle$ in $T(m)$ let a_b be a

copy of $a(m)$ and $\theta = \{v_1/t_1, .., v_n/t_n\}$. If Prolog with depth-bound h succeeds on goal $a_b\theta$ with the set of answer substitutions Θ_b then for each θ_b in Θ_b and for each v/t in θ_b if v corresponds to a #type in m then replace v in a_b by t otherwise replace v in a_b by v_k where $k = hash(t)$ and add v to InTerms if v corresponds to -type. Add $\overline{a_b}$ to $\bot_i$.

6. Goto step 4.

In the following we describe Progol's A^*-like algorithm for finding clause with maximal compression. First we define some auxiliary functions which are used in this algorithm.

Definition 98 Auxiliary functions. *Let the examples E be a set of Horn clauses. Let $h, i, B, e, M, \bot_i$ be as in Definition 43 and let C, k, θ be as in Definition 45.*

$$d'(v) = \begin{cases} 0 & \textit{if there is no -type variable in the head of } \bot_i \\ 0 & \textit{if } v \textit{ is -type in the head of } \bot_i \\ \infty & \textit{if } v \textit{ is not in } \bot_i \\ (min_{u \in U_v} d'(u)) + 1 & \textit{otherwise} \end{cases}$$

where U_v are the -type variables in atoms in the body of C which contain +type occurrences of v. Below state s has the form $\langle C, \theta, k\rangle$. c is a user-defined parameter for the maximal clause body length. $|S|$ denotes the cardinality of any set S.

$$\begin{aligned} p_s &= |\{e : e \in E \textit{ and } B \wedge C \wedge \overline{e} \vdash_h \Box\}| \\ n_s &= |\{e : e \in E \textit{ and } B \wedge C \wedge e \vdash_h \Box\}| \\ c_s &= |C| - 1 \\ V_s &= \{v : u/v \in \theta \textit{ and } u \textit{ in body of } C\} \\ h_s &= min_{v \in V_s} d'(v) \\ g_s &= p_s - (c_s + h_s) \\ f_s &= g_s - n_s \end{aligned}$$

best(S) is a state $s \in S$ which has $c_s \leq c$ and for which there does not exist $s' \in S$ for which $f_{s'} > f_s$.

$$prune(s) \;=\; \begin{cases} true & \textit{if } n_s = 0 \textit{ and } f_s > 0 \\ true & \textit{if } g_s \leq 0 \\ true & \textit{if } c_s \geq c \\ false & \textit{otherwise} \end{cases}$$

$$terminated(S, S') \;=\; \begin{cases} true & \textit{if } s = best(S), n_s = 0, f_s > 0 \textit{ and} \\ & \textit{for each } s' \textit{ in } S' \textit{ it is the case that } f_s \geq g_{s'} \\ false & \textit{otherwise} \end{cases}$$

Algorithm A.2 A^*-like algorithm for searching the lattice defined by $\bot_i$.

1. Given $h, B, e, \bot_i$ as in Definition 43.

2. Let Open = $\{\langle\Box, \emptyset, 1\rangle\}$ and Closed = $\emptyset$.

3. Let s = best(Open) and Open = Open$-\{s\}$.

4. Let Closed = Closed $\cup\{s\}$.

5. If prune(s) goto 7.

6. Let Open = (Open$\cup\rho(s)$)$-$Closed.

7. If terminated(Closed,Open) then return best(Closed).

8. If Open = $\emptyset$ then print 'no compression' and return $\langle e, \emptyset, 1\rangle$.

9. Goto 3.

In the following we describe Progol's cover set algorithm.

Definition 99 Unflattening. *Let $C = h \leftarrow X, Y$ be a definite clause in which $X = (s_1 = t_1, .., s_n = t_n)$ is a conjunction of atoms with predicate symbol '$=$' and Y is a conjunction of atoms with predicate symbols other than '$=$'. The clause $C' = h' \leftarrow Y'$ is called the unflattening of C if and only if C' is derived from C by successively resolving away each $s_i = t_i$ in X with the clause ($U = U \leftarrow$).*

Algorithm A.3 Progol's cover set algorithm

```
1. $h, i, B, M$ are given as in Definition 43 and $E$ is the subset of $B$ corresponding to atoms in modeh declarations in $M$.
2. If $E = \emptyset$ then return $B$.
3. Let $e$ be the first example in $E$.
4. Construct $\perp_i$ for $e$ using Algorithm A.1
5. Construct state $s$ from $\perp_i$ using Algorithm A.2
6. Let $C'$ be the unflattening of $C(s)$ (Definition 99).
7. Let $B = B \cup C'$.
8. Let $E' = \{e : e \in E \text{ and } B \wedge \overline{e} \vdash_h \Box\}$.
9. Let $E = E - E'$.
10. Goto 2.
```

Appendix B

Examples of logic-based learning using genetic search

In this section we describe some of logic-based learning systems which use a genetic search. In particular features which are relevant to the implementation and evaluation of algorithms in this thesis (e.g. representation, operators and fitness function) are discussed in this section.

GA-SMART

The lack of a proper (binary) representation, and consequently difficulties for definition and implementation of genetic operators, has been a main problem for applying standard GAs in first-order domain. GA-SMART [GS92] was the first relational learning system which tackled this problem by restricting concept description language and introducing a language template. A template in GA-SMART is a fixed length CNF formula which must be defined by the user. Mapping a formula into bit-string is done by setting the corresponding bits to represent the occurrences of predicates in the formula. An example of a language template and mapping a formula to a bit-string are shown in Figure B.1.

Genetic search in GA-SMART is guided by a fitness function which combines completeness and consistency together with simplicity of the relation measured by the number of literals in the formula. The following is the fitness function used in GA-SMART:

$$f(\varphi) = \alpha * [m^+(\varphi)/(M^+ + \beta * m^-(\varphi))] + \gamma * (1 - n(\varphi)/n(\Lambda)) \qquad \text{(B.1)}$$

$$\begin{aligned}
\Lambda = \ & [\mathit{black}(x) \vee \mathit{grey}(x) \vee \mathit{dotted}(x)] \wedge [\mathit{black}(y) \vee \mathit{grey}(y) \vee \mathit{dotted}(y)] \wedge \\
& [\mathit{black}(z) \vee \mathit{grey}(z) \vee \mathit{dotted}(z)] \wedge [\mathit{black}(w) \vee \mathit{grey}(w) \vee \mathit{dotted}(w)] \wedge \\
& [\mathit{next\_right}(x, y) \vee \mathit{not\_next\_right}(x, y)] \wedge [\mathit{next\_right}(x, z) \vee \mathit{not\_next\_right}(x, z)] \wedge \\
& [\mathit{adj}(x, w) \vee \mathit{not\_next\_right}(x, w)] \wedge [\mathit{next\_right}(y, z) \vee \mathit{not\_next\_right}(y, z)] \wedge \\
& [\mathit{next\_right}(y, w) \vee \mathit{not\_next\_right}(y, w)] \wedge [\mathit{next\_right}(z, w) \vee \mathit{not\_next\_right}(z, w)] \\
\varphi = \ & \mathit{black}(x) \wedge [\mathit{grey}(y) \vee \mathit{dotted}(y)] \wedge \mathit{next\_right}(x, y) \wedge \mathit{next\_right}(z, w)
\end{aligned}$$

(a) A language template Λ and a formula φ

1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0

(b) Mapping formula φ to a bit-string

Figure B.1: An example of a language template and mapping a formula to a bit-string in GA-SMART [GS92].

In this formula M^+ is the number of positive instances of the target concept in the training set, $m^+(\varphi)$ is the number of positive instances verifying a formula φ, $m^-(\varphi)$ is the number of negative instances, and $n(\varphi)$ the number of literals occurring in φ. The first term of fitness function captures completeness and consistency and the second term evaluates the simplicity of the formula by comparing the number of literals occurring in φ to the global number of literals defined in template Λ.

Other novelty of the system is that it can use generalising and specialising operators as well as conventional uniform and one-point crossover operators. These genetic operators randomly generalise and specialise a formula by bitwise ANDing and ORing of parent strings. For this purpose a subset of template randomly selected and the corresponding bits of parents are ANDed or ORed together. Figure B.2 shows an example of generalising crossover operator in GA-SMART. The probabilities for uniform, two-point, specialising and generalising crossover operators are as follows:

$$p_u = (1 - a * f) * b \tag{B.2}$$

$$p_{2pt} = (1 - a * f) * (1 - b) \tag{B.3}$$

$$p_s = a * f * r \tag{B.4}$$

$$p_g = a * f * (1 - r) \tag{B.5}$$

In these equations, a and b are tunable parameters, $f = [f(s_1) + f(s_2)]/2$ is the mean value of the fitness of two parental strings s_1 and s_2, and $r = [(m^+(s1) + m^-(s_1) + m^+(s_2) + m^-(s_2)]/[(M^+ + M^-) * 2]$ is the mean value of the ratio between the number of instances covered by $\varphi(s_1)$ and $\varphi(s_2)$, respectively, and the global number of instances in the training set.

$$\Delta = [black(w) \vee grey(w) \vee dotted(w)]$$

(a) Randomly selected subset of the template

1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
1	1	0	1	0	1	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0

(b) Parents

1	0	0	0	1	1	0	0	0	**1**	**1**	**0**	1	0	0	0	0	0	0	0	0	0	1	0
1	1	0	1	0	1	1	0	0	**1**	**1**	**0**	0	1	0	1	0	0	0	0	0	0	0	0

(c) Offsprings

Figure B.2: An example of generalising crossover in GA-SMART [GS92]. First Δ, a subset of the template is randomly generated then the corresponding bits of parents are ORed together.

GA-SMART can learn several concepts at the same time by using a distributed genetic algorithm in which each sub-population can evolve a different concept. This system uses several standard methods in GAs such as fitness scaling, crowding factors as well as migration of individuals between sub-population.

REGAL [GN96] and G-NET [AGLS98] closely follow the same idea of GA-SMART and employ user-defined templates for mapping first-order clauses into bit strings. These systems therefore need to adopt a non-standard first-order representation based on a template which is a conjunction of internally disjunctive predicates. The main problem of this approach is that the number of conjuncts grows combinatorially with the number of predicates. The template, therefore, can be very large in some circumstances and difficulties related to defining the template by the user is also a major disadvantage of this approach.

DOGMA

DOGMA [Hek98] is a learning system which uses GAs for learning first-order concepts. Like GA-SMART, this system uses language templates for encoding first-order logic into binary strings and mapping a formula to a binary string is done by setting the corresponding bits of the template. An example of a language template in DOGMA and mapping a formula to a bit-string are shown in figure B.3. In this template, the symbol * may be used to collapse a set of values.

Genetic operators in DOGMA are defined at two different levels: chromosomes level

$\Lambda = size(x, [s, m, l]) \wedge shape(x, [sq, tr, *]) \wedge shape(y, [sq, tr, *]) \wedge ontop(x, y, [yes, no])$
$\varphi = size(x, [s, m]) \wedge shape(y, [tr]) \wedge ontop(x, y, [yes])$

(a) A language template Λ and a formula φ

1	1	0	1	1	1	0	1	0	1	0

(b) Mapping formula φ to a bit-string

Figure B.3: An example of a language template (a) and mapping a formula to a bit-string (b) in DOGMA [Hek98].

and families level. Families are built by selecting useful chromosomes of different species. Speciation is done by background seeding operator which randomly selects sub-structures from background knowledge and encode them into bit-strings. Hence, chromosomes are divided into species according the part of background knowledge which they may use. The purpose of speciation is to enhance diversity and to separate different kinds of rules. This system uses two operator for manipulating families. These operators are *break* and *join* which randomly split or merge families into new families. Evaluation and selection are also done at family level. DOGMA combines Minimum Description Length (MDL) [QR89] with Information Gain(IG) [Qui90] and uses the minimum of these as the fitness value. Other features including crossover operators are similar to GA-SMART.

DOGMA has suffers from the same disadvantages related to using a template as mentioned about GA-SMART.

SIAO1

SIAO1 [SAK95] is a learning system which combines the covering-set algorithm of the learning system AQ [MMHL86] with a GA. The covering-set algorithm, which have been used in many of inductive learning systems, starts by selecting a positive example as a seed and then continues by generalising the seed example. The role of GA in SIAO1 is to search for the best generalisation of the seed example according to a rule-evaluation criterion.

SIAO1 uses a direct mapping for representing first-order rules which means that no binary encoding is required. Crossover is done by randomly selecting a site and then

Pyramid	X	Colour	X	yellow	Supports	Y	X	Ø	Ø	Ø
Ø	Ø	Colour	Y	yellow	Supports	c	d	Length	d	7

(a)

Pyramid	X	Colour	X	yellow	Supports	c	d	Length	d	7
Ø	Ø	Colour	Y	yellow	Supports	Y	X	Ø	Ø	Ø

(b)

Figure B.4: An example of crossover operator in SIAO1 [SAK95]. Predicates and arguments after crossover point in parental rules (a) are exchanged and two offspring (b) are generated.

swapping the predicates and arguments between parental rules. Figure B.4 shows an example of crossover operators in SIAO1. Mutation in this system is done by randomly selecting a predicate or variable and then generalising it according to a hierarchical background knowledge which is known for the problem. This background knowledge includes a hierarchy of concepts and predicates. Hence, this system uses a simple form of background knowledge. Fitness function in SIAO1 combines completeness and consistency together with syntactical generality (which can be measured by the number of variables in the formula or the size of disjunctions and intervals) and user preference for presence of some relations in the concept.

The genetic operators in SIAO1 are limited because of the direct representation which could lead to a low diversity population. For example in crossover operator, two genes at the same position must represent the same predicate. The other problem of SIAO1 is due to its unusual genetic algorithm which starts with only one individual and therefore the genetic search is sensitive to the choice of the seed example.

GLPS

GLPS [LW95] is a learning system based on Genetic Programming (GP) [Koz91] where hierarchical representations are used rather than fixed length bit-strings. However, unlike most GP systems which use Lisp expressions as the representation language, GLPS uses logic programs. This system evolves a randomly generated forest of AND-OR trees each corresponding to a rule of the logic program. Figure B.5 shows a logic program and corresponding AND-OR trees. Crossover point is determined by a randomly generated set of numbers. According to this set, the crossover operator

$C1 : cup(?x) : -insulate\_heat(?x), stable(?x), liftable(?x).$
$C2 : cup(?x) : -paper\_cup(?x).$
$C3 : stable(?x) : -bottom(?x, ?b), flat(?b).$
$C4 : stable(?x) : -bottom(?x, ?b), concave(?b).$
$C5 : stable(?x) : -has\_support(?x).$
$C6 : liftable(?x) : -has(?x, ?y), handle(?y).$
$C7 : liftable(?x) : -small(?x), made\_from(?x, ?y), low\_density(?y).$

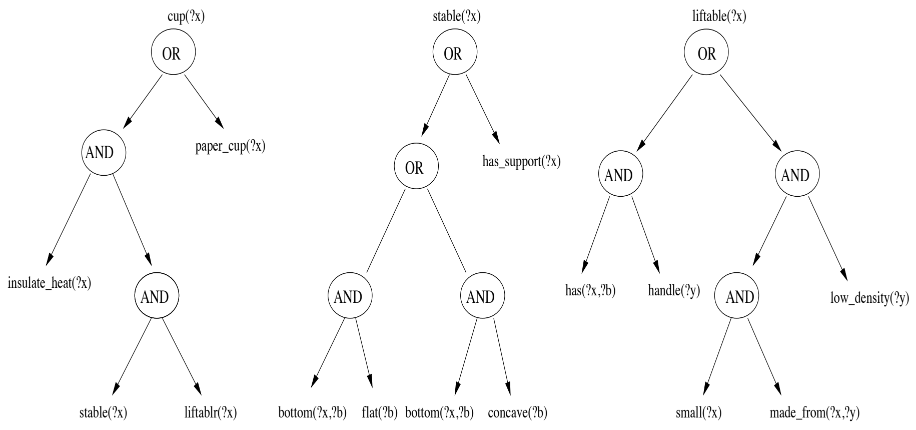


Figure B.5: Examples of program trees in GLPS [LW95].

exchanges a whole logic program, some rules, clauses or literals between parents. One important feature of this system is that unlike conventional GP, the induced logic programs are syntactically valid and there is no need for GP closure assumption. GLPS uses GP as the main mechanism for inducing logic programs and therefore it cannot benefit from background knowledge during the learning process. All other features are similar to conventional GP.

LOGENPRO [WL97] is a generalisation and extension of GLPS which can learn programs not only in Prolog but also in different programming languages including LISP and Fuzzy Prolog. All other features are similar to GLPS and it shares the same advantages and disadvantages.

STEPS

STEPS [KGC99] is a learning system which can induce higher-order concepts. This system uses Escher programming language [Llo95] to represent examples and hypotheses. Escher is a combination of logic programming and functional programming. This system, therefore, can represent highly structured concepts.

STEPS uses Strongly Typed Genetic Programming (STGP) [Mon93] to evolve higher-order concepts in the form of program trees. In this system, crossover operators are modified to preserve type consistency and variable consistency of STGP. This system also uses special kinds of mutation operators which randomly add or drop disjuncts and conjuncts to the program tree. Other features are similar to conventional GP systems, in particular those based on STGP.