Search Improvements in Multirelational Learning

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieurin
(Dr.-Ing.)

angenommen durch der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von	M. Sc. María de Lourdes Peña Castillo
geboren am	02. Mai 1974 in Mexico, D.F.

Gutachter:	Prof. Dr. Stefan Wrobel
	Prof. Dr. Stefan Kramer
	Prof. Dr. Andreas Nürnberger

Magdeburg, den 7. June 2004

Abstract

In this thesis we lay the foundations to develop multirelational learning systems which can cope better with the challenges posed by structural and topological domains. Even though many interesting application domains contain structural or topological data, current multirelational systems have difficulties dealing with the complexity of the search space of theses domains, their indeterminacy, and the presence of non-discriminating relations. In this work, we describe *macro-operators* which are a formal method to reduce the search space explored in structural or topological domains and can also be used to alleviate the myopia of greedy systems. We also explore parallel search based on stochastically selected examples to reduce the instability of example-driven learning. As a third contribution, we present *active inductive learning* as an approach to improve the efficiency of the instance space exploration.

Zusammenfassung

Diese Dissertation legt die Grundlage zur Entwicklung multirelationaler Lernverfahren, die strukturelle und topologische Anwendungsbereiche erfolgreich bearbeiten können. Obwohl viele gängige Anwendungsbereiche strukturelle und topologische Daten enthalten, ist es für herkommliche multirelationale Lernverfahren schwierig, diese Lernaufgabe zu bewältigen. Das liegt an der Komplexität ihres Suchraums, an ihrer Unbestimmtheit und an den vorliegenden nichtdiskriminierenden Relationen. Diese Arbeit stellt die Methoden *macro-operators*, *Parallel-Suche* und *aktives induktives Lernen* vor. Macro-operators sind eine formale Methode, um den Suchraum in den erwähnten Anwendungsbereichen einzuschränken. Zudem sind macro-operators geeignet zur Verringerung der Kurzsichtigkeit von auf gieriger Suche basierenden Systemen. Durch die Anwendung der Parallel-Suche, die auf zufällig ausgewählten Beispielen basiert, kann die Stabilität von example-driven Verfahren verbessert werden. Aktives induktives Lernen ist eine Methode, die eine effizientere Erforschung des Instanzenraums ermöglicht.

Acknowledgments

Although a Ph.D. dissertation is usually seen as the result of a long, dedicated, individual research work, there are many people besides the author who contribute to the completion of it. In my case, I want to thank in this page all those who directly or indirectly help me to finish this work.

Specially, I would like to thank

- Stefan Wrobel for his guidance, support and supervision.
- Stefan Kramer and Andreas Nürnberger for reviewing this thesis.
- The members of the Institute for Knowledge and Language Processing for providing a friendly working environment.
- All the friends who made my stay in Magdeburg a lot more enjoyable and the departure a lot more difficult, I feel really honour to have the fortune of sharing with you so many memories.
- My family for always believing in me and for their unconditional love.
- My partner in this adventure, Oscar Meruvia, for all we have constructed together, being who he is, and his involvement in my work.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

The question whether computers are able to learn has been a perennial topic in artificial intelligence research. As far back as 1959, Samuel [96] worked on a checkers program that improved its performance through learning. Machine learning is the branch of artificial intelligence (AI) which deals with the construction of systems which are able to learn. *Being able to learn* means that the system is capable of improving with experience, or of the autonomous acquisition and integration of knowledge. Multirelational learning or inductive logic programming (ILP) [33, 71, 76, 103] is a subfield of machine learning concerned with learning concept definitions from examples using a *first-order* representation.

Multirelational learning has been applied to several real-world problems[1] such as finite element mesh design [26], predicting the mutagenicity of chemical compounds [100], classification of river water quality [7], detection of traffic problems [32], and protein secondary structure prediction [74]. Despite these successes, ILP systems still have difficulties dealing with application domains containing *structural* or *topological* data. *Structural data* contains information about the parts, arrangement and composition of complex entities or objects; e.g., information about the components of a mechanical structure or about the hierarchy of a company. *Topological data* contains information about geometrical and spatial relations among entities or objects; e.g., the relative position of a piece in a board game with respect to other pieces on the board or the position of a robot in a room. Many interesting application domains contain structural or topological data. In this thesis, we address the question: how to efficiently learn in domains with structural/topological information. Specifically, we lay the foundations to develop ILP systems which can better cope with the challenges posed by domains with structural or topological data. These challenges are the *combinatorial nature* of structural/topological domains, their *indeterminacy*, and the presence of *non-discriminating* relations.

[1]For an overview of ILP applications the reader is referred to [8, 28].

In the next section we explain why a first-order data representation is needed. After that, we illustrate the challenges posed by structural/topological domains. Section 1.3 enumerates the contributions of this thesis, and Section 1.4 describes this thesis structure.

1.1 Why a Relational Data Representation?

Machine learning approaches (e.g., neural networks, Bayesian learning, reinforcement learning, genetic algorithms, decision tree learning, multirelational learning, and kernel methods) differ from each other in the way knowledge and data are represented. A main distinction is drawn between *propositional* (also called *attribute-value*) and *relational* (also called *first-order*) representations. In a propositional representation, the data contains attributes (or features) and their values, and an example is usually described as a vector of values of fixed size. On the other hand, in a relational representation, the data contains not only information about the value of the attributes but also about relations among elements in the domain of discourse, and an example is usually described as a set of tuples of several relations.

Due to the simplicity of their data representation, propositional algorithms are usually more efficient than relational ones. Because of this efficiency, several researchers [56] have developed methods to transform relational learning problems into a propositional representation and then solve them using a propositional algorithm. Such a transformation is called *propositionalization*. In database terminology, propositionalization means to store in one single table the information contained in multiple tables of a relational database. However, as described in [105], there are two problems associated with this process: redundancy and information loss. What is more, De Raedt [17] has shown that the complete transformation of non-trivial multirelational problems to an attribute-value representation is computationally too expensive to be applied in practice.

In addition, a first-order representation has two main advantages over an attribute-value representation. First, the relational representation provides with a more expressive description language, and second, the learning results are more understandable and general. As an example, consider a learning problem from the games domain: to predict whether a given unknown tile in a Minesweeper[2] board (see Figure 1.1) contains a mine or not. In an attribute-value representation, we can choose to have a binary target attribute indicating whether a square in the board contains a mine (e.g., **tile4IsMine** is true if the top right square

[2]Minesweeper is a common one-person computer game. More about this game can be found in Chapter 8.

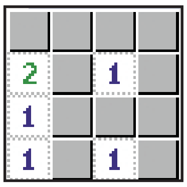


Figure 1.1: A Minesweeper board

boardID	**tile4IsMine**	**tile1**	**tile2**	**tile3**	**tile4**	**tile5**	...	**tile16**
1	false	null	null	null	null	2	...	null
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 1.1: An attribute-value representation of Minesweeper

minesAround(tile5,2)	isMine(tile1)
minesAround(tile7,1)	not(isMine(tile3))
minesAround(tile9,1)	not(isMine(tile4))
⋮	⋮
neighbourOf(tile5,tile1)	
neighbourOf(tile5,tile9)	
⋮	

Table 1.2: A relational representation of Minesweeper

is a mine), and we can also have integer attributes indicating the number of mines around a square (see Table 1.1). Since in Minesweeper the neighborhood relation between tiles is very important, we need to introduce this information in our propositional representation. One way to do it is to provide a set of eight attributes for each tile to indicate its neighbours. This representation is not only too cumbersome but also dependent on the size of the board and contains null values for several attributes. Besides that, the learning result would be a collection of very specific rules such as:

$$\text{if } (\textbf{tile4} = 0) \text{ and } (\textbf{neighbour}_{4,1} = \text{tile3}) \text{ then } (\textbf{tile3IsMine} = \text{false}).$$

In contrast, in a first-order representation we can have several relations such as **minesAround**, **neighbourOf**, and **isMine** and represent the examples and the information on the board as shown in Table 1.2. With a relational representation, the following rule, for instance, could be obtained:

if **minesAround(A,0)** and **neighbourOf(A,B)** then **not(isMine(B))**.

This rule is more general than our propositional one because it contains variables and relations. In addition, a first-order representation is independent of the size of the board and allows us to include in the learning process knowledge about relations among elements in the domain. That is, in our Minesweeper example, we can express the spatial relations among the tiles and the arithmetic relations among the numbers appearing on the tiles.

1.2 Challenges of Structural/Topological Domains

To illustrate why structural/topological domains are challenging for ILP Systems, let us take the domain of hand drawn sketches. The learning task is to find rules to classify hand drawn sketches according to the object they represent; e.g., chair, wheel, table, etc. Since the sketches are composed of simple geometrical forms (e.g., circle, square and triangle), the domain has structural relations such as **partOf**. For example, a circle is part of a wheel sketch. In addition, there are topological relations describing the placing of each simple form (also called stroke) with respect to other strokes in the sketch; e.g., **overlap**, **near**, and **insideOf**. Finally, there are also predicates describing properties of the strokes such as **shape**, **size**, and **orientation**. In this domain, the following rule could be used to classify hand drawn sketches of wheels.

if **partOf(Sketch,Stroke1)** and **shapeOf(Stroke1, circle)** and **insideOf(Stroke1,Stroke2)** and **shapeOf(Stroke2, circle)** then **wheel(Sketch)**.

We explain now the challenges imposed by structural/topological domains.

1. **Complexity of the search space**

   Predicates denoting structural or topological relations introduce references to new elements in the condition part of the rules. For example, relations such as **partOf** and **insideOf** introduce a new reference to a stroke in the rule. When a reference to a new stroke (e.g., **Stroke1** in the rule above) has been made in a rule all predicates characterizing this stroke can be included in the rule too. Some of these predicates may denote as well structural or topological relations and introduce new references to other elements in the domain (e.g., **Stroke2**), which in turn originates that predicates about these other elements have to be taken into account for inclusion in the rule, and so on.

   The set of all rules that can be constructed using the predicates denoting relations among elements in the domain or characterizing those elements is called *hypothesis space*. By depicting the hypothesis space as a search tree, one can visualize that every new reference to an element creates new branches in the search tree. Each new branch in the search tree increases exponentially the size of the hypothesis space; i.e., the number of rules we have to look at.

Depending on the number of elements and relations in the domain, the search tree can rapidly become so huge that it is not possible to look efficiently at all the rules to find the "best" ones. In this case, one alternative is to prune the tree; i.e., to leave out some branches. However, by pruning the search tree, we may remove the rules we are looking for. To avoid this, what is needed is an approach to reduce the size of the search tree with the following properties.

a) **Safe**: Only rules which cannot be selected as best clauses should be discarded.

b) **General**: The approach should be suitable for every structural/topological domain.

c) **Effective**: The search space should be substantially reduced.

d) **User-friendly**: The approach should not imply extra work for the user.

However there is no such approach in ILP.

2. **Non-discriminating relations**

Usually, relations that introduce into a rule new references to elements in the domain do not help to distinguish among examples belonging to different classes. For instance, the relation **partOf** does not help to differentiate between sketches of wheels and sketches of chairs because both sketches are formed by strokes. We call these relations *non-discriminating*.

Suppose we have a search algorithm, which evaluates all the relations that can be included in the condition part of a rule at one level of the search tree, and then, selects the one with the highest evaluation value before looking at the next level of the search tree. Such a search algorithm is called *hill-climbing search*. Because **partOf** does not differentiate between examples belonging to different classes, it is evaluated by hill-climbing search as unimportant or useless, and it is not selected to construct a rule. Thus, hill-climbing search may end up with a non-optimal rule. In this case, the algorithm cannot "see" that **partOf** combined with other relations is important and it is said to suffer from *myopia*. Currently, there are ILP approaches which reduce the myopia of hill-climbing; however, they are either general or user-friendly but not both.

3. **Instability**

   Several ILP systems use one or a few positive examples to guide the search in the hypothesis space. This approach is called *example-driven* or *data-driven* and is quite effective to learn complex rules (such as the rules usually needed in structural/topological domains). However, this approach has the drawback that since only one or a few individual examples are selected as a basis to search for a rule in each round, the choice of examples can have a significant effect on the quality and contents of the rules obtained. That means that the user cannot rely on obtaining identical, or at least identically performing rules in different runs with the same examples; i.e., the learning system is *instable*.

   Although it is well known that example-driven systems potentially exhibit the instability described above, their instability has not been considered in great detail in the literature.

4. **Inefficient exploration of the instance space**

   Since in many games structural and topological relations constitute an important part of the relations in the domain, games have been a frequent application domain in our research. In this way, we realized that games introduce an extra challenge for ILP which is the amount of examples (i.e., thousands) required to obtain information about most of the possible game situations. Considering thousands of examples when evaluating a rule slows down the learning process because the rule has to be applied to every single example.

   In addition, since in games there are rare or exceptional cases which have to be considered by the set of rules obtained, random sampling is not an optimal solution because it may result that no instances of some rare case are present in the examples (which could prevent the system from learning a complete set of rules). For example, in chess one may have a lot of examples about the opening moves but very few examples about end-game moves of king-rook versus king-knight, and for a learning system it would be optimal to have only some instances of opening moves and all the examples available for end-game moves.

1.3 Thesis Contributions

The main motivation behind this work is to lay the foundations to develop multirelational learning systems capable to better deal with the challenges posed by structural/topological application domains. Specifically in this thesis we proposed the following solutions to deal with the problems described above.

- **Problem: Complexity of the search space.**
  Thesis contribution: Macro-operators.

  We propose *macro-operators* as a formal approach to reduce the hypothesis space searched by a system in domains with topological/structural data. Macro-operators are general, safe, effective and user-friendly. We also elaborate a formal theory which supports the use of macro-operators, and provide correct and complete algorithms for their generation.

- **Problem: Non-discriminating Relations.**
  Thesis contribution: Macro-operators.

  We introduce macro-operators also as an approach to allow ILP learning systems to more accurately assess the relevance of structural/topological relations to construct adequate rules. Macro-operators do not imply extra work for the user and can be applied to every structural/topological domain; i.e., they are general and user-friendly.

- **Problem: Instability.**
  Thesis contribution: Parallel Search.

  We show that by performing independent parallel searches on several stochastically selected examples, the instability of example-driven learning is reduced. Parallel search is combined with exhaustive and hill-climbing search.

- **Problem: Inefficient exploration of the instance space.**
  Thesis contribution: Active Inductive Learning.

  We introduce *active inductive learning* to deal with the instance space exploration problem. Active inductive learning consists in integrating *active learning* [15] in a multirelational learning framework. Basically, a system uses the knowledge it obtains during learning to suggest explorations of unknown portions of the instance space and uses this exploration to validate the rules already obtained.

To empirically evaluate these contributions, we developed Mio, an example-driven multirelational learning system.

1.4 Thesis Outline

This thesis is organized as follows. Chapter 2 introduces logic programming notions relevant to ILP and Chapter 3 examines learning and search issues concerning multirelational learning. The goal of these two chapters is to provide all the background material required to understand the remaining chapters.

Chapter 4 describes in detail a basic approach for ILP which is modified and improved through the thesis.

Chapter 5 elaborates the formal theory behind macro-operators, provides algorithms for their generation, identifies the required properties of the application domain that allow their use, and introduces Mio, our multirelational learning system. This chapter is partially based on the paper *Macro-operators in Multirelational Learning: a Search-Space Reduction Technique* [79] presented in 2002 at the European Conference on Machine Learning.

In Chapter 6, the use of macros to overcome the myopia of hill-climbing search is explained and compared with previous approaches to solve the shortsightedness of greedy algorithms. Part of the work described in this chaper is contained in the paper *A Comparative Study on Methods for Reducing Myopia of Hill-climbing Search in Multirelational Learning* [83] accepted for presentation at the International Conference on Machine Learning, 2004.

Chapter 7 is dedicated to discuss in detail parallel search. This chapter is partially based on the paper *On the Stability of Example-Driven Learning Systems: a Case Study in Multirelational Learning* [81] which won a Best Conference Paper Award at the Mexican International Conference on Artificial Intelligence 2002.

In Chapters 5 to 7 the reader will find sections devoted to related work and empirical evaluations of each contribution.

Chapter 8 introduces the task of learning a playing strategy for Minesweeper, a widely distributed one-person computer game, which is taken as an instance of a highly structural/topological application domain. In addition, this chapter discusses active inductive learning which allows for an efficient exploration of the instance space and illustrates how the use of macro-operators and active inductive learning allowed Mio to learn a playing strategy for this game. Part of the work described in Chapter 8 is contained in the paper *Learning Minesweeper with Multirelational Learning* [82] presented at the International Joint Conference on Artificial Intelligence in 2003.

Conclusions and pointers to future work are given in Chapter 9.

2 Logic Programming for ILP

This chapter discusses logic programming notions relevant for multirelational learning and its main goal is to define the terms used throughout the thesis. As usual in logic, we deal separately with the *syntax* (the rules to construct well-formed structures admitted by the grammar of the formal language) and the *semantics* (the meaning assigned to those structures) of first-order logic. The definitions here provided are mostly taken from [61] and [76]. For this chapter, it is assumed the reader has some logic background. A complete introduction to logic programming can be found in [61].

2.1 Syntax

The first thing we need to construct well-formed syntactical structures admitted by the grammar of a formal language is an *alphabet*.

Definition 2.1 – Alphabet. *An* alphabet *consists of the following classes of symbols:*

1. *A set of constants.*
2. *A non-empty set of variables.*
3. *A set of function symbols. Each function symbol has a natural number (its* arity*) assigned to it.*
4. *A non-empty set of predicate symbols. Each predicate symbol has a natural number (its* arity*) assigned to it.*
5. *Five connectives:* $\neg$, $\vee$, $\wedge$, $\rightarrow$, *and* $\leftrightarrow$.
6. *Two quantifiers:* $\exists$ *and* $\forall$.
7. *Three punctuation symbols: '(', ')' and ','.*

Definition 2.2 – Arity. *The* arity *of a function or predicate symbol gives the number of arguments the function or predicate symbol has.*

For instance, the mathematical function $f(x, y) = x + y$ has two arguments and its arity is thus two. Predicates and functions are commonly described by their name followed by their arity, e.g., $f/2$. In general, a function or predicate symbol of arity n is called an n-ary function or predicate. In this thesis, we adopt Prolog conventions for naming constants, variables, function and predicate symbols.

Now we define the two well-formed syntactic structures (*terms* and *formulas*) that can be constructed with an alphabet.

Definition 2.3 – Term. *A* term *is defined as follows:*

1. *A variable is a term.*
2. *A constant is a term.*
3. *If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*

Definition 2.4 – Formula. Well-formed formulas *(or just* formulas*) are defined as follows:*

1. *If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula, called an* atom.
2. *If ϕ and ψ are formulas, then $\neg\phi$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $(\phi \rightarrow \psi)$, and $(\phi \leftrightarrow \psi)$ are formulas.*
3. *If ϕ is a formula and X is a variable, then $(\forall X \phi)$ and $(\exists X \phi)$ are formulas.*

Definition 2.5 – Ground Term. *A* ground term *(respectively* ground formula*) is a term (formula) which does not contain any variables.*

From an alphabet, one can construct an infinite number of formulas, and all these formulas form a *language*.

Definition 2.6 – First-order Language. *The* first-order language *given by an alphabet is the set of all formulas which can be constructed from the symbols of the alphabet.*

Logic programs use a restricted first-order language where the only class of formulas allowed are *clauses*. This subset of first-order logic is called *clausal logic*.

Definition 2.7 – Literal. *A* literal *is an atom or the negation of an atom. A* positive literal *is an atom, a* negative literal *is the negation of an atom.*

Definition 2.8 – Clause. *A* clause *is a formula which consists of a finite disjunction of zero or more literals. That is, it has the form $\forall X_1 \ldots \forall X_s (l_1 \vee \ldots \vee l_m)$ where l_i is a literal and $X_1, \ldots, X_s$ are all the variables occurring in $l_1 \vee \ldots \vee l_m$.*

Definition 2.9 – Function-free. *A clause is* function free*, if it does not contain function symbols of arity 1 or more. A set of clauses is* function-free *if all its members are function-free.*

Definition 2.10 – Clausal Language. *The* clausal language *given by an alphabet is the set of all clauses which can be constructed from the symbols of the alphabet.*

In logic programming, a special clausal notation has been adopted. In this notation the clause

$$\forall X_1 \ldots \forall X_s (a_1 \vee \ldots \vee a_k \vee \neg g_1 \vee \ldots \vee \neg g_n)$$

where $a_1, \ldots, a_k, g_1, \ldots, g_n$ are atoms and $X_1, \ldots X_s$ are all the variables occurring in these atoms, is denoted by

$$a_1, \ldots, a_k \leftarrow g_1, \ldots, g_n$$

In this clausal notation, all variables are assumed to be universally quantified, the commas in the condition part $g_1, \ldots, g_n$ denote conjunction, and the commas in the conclusion part $a_1, \ldots, a_k$ indicate disjunction.

A clausal language is usually further restricted to *Horn clauses*. Although there is a loss of expressive power by using Horn clauses, it is compensated by a gain in tractability: sets of Horn clauses are practically and theoretically easier to handle than sets of general clauses. In fact, the programming language Prolog uses Horn clauses and most work in ILP is only concerned with Horn clauses.

Definition 2.11 – Definite Program Clause. *A* definite program clause *is a clause containing one atom in its conclusion part and zero or more literals in its condition part. That is, it has the form $a \leftarrow g_1, \ldots, g_n$. a is called the* head *and $g_1, \ldots, g_n$ the* body *of the program clause.*

Definition 2.12 – Unit Clause. *A* unit clause *or* fact *is a definite program clause with an empty body (i.e., $a \leftarrow$).*

Definition 2.13 – Definite Goal. *A* definite goal *is a clause which has an empty head (i.e., $\leftarrow g_1, \ldots, g_n$). Each g_i is called a* subgoal *of the goal.*

Definition 2.14 – Empty Clause. *The* empty clause*, denoted $\square$, is the clause with empty head and empty body. This clause is to be understood as a contradiction and is also considered to be a goal.*

Definition 2.15 – Horn Clause. *A* Horn clause *is either a definite program clause or a definite goal.*

$$\overbrace{\overbrace{\text{no\_payment\_due(Person)}}^{head} \leftarrow \overbrace{\underbrace{\underbrace{\text{enrolled}}_{predicate}(\underbrace{\text{Person, School, N}}_{arguments})}_{literal}\text{, geq(N, 12)}}^{body}}^{Horn\quad clause}.$$

Figure 2.1: An example of a Horn clause

Figure 2.1 illustrates a Horn clause and its parts. For simplicity, we refer often to Horn clauses as clauses. Horn clauses are usually ended with a point.

Definition 2.16 – Definite Program. *A* definite program *is a finite set of definite program clauses.*

Definition 2.17 – Predicate Definition. *In a definite program, the set of all program clauses with the same predicate symbol* **p** *(and arity) in their heads is called the* definition *of* **p**.

For example, the following three clauses form a predicate definition of the predicate symbol **no_payment_due**/1.

no_payment_due(Person) ← **unemployed(Person).**
no_payment_due(Person) ← **filed_for_bankruptcy(Person).**
no_payment_due(Person) ← **enrolled(Person,School,N),geq(N,12).**

In the same way as we have clausal languages, we also have *Horn languages.*

Definition 2.18 – Horn Language. *The* Horn language *given by an alphabet is the set of all Horn clauses which can be constructed from the symbols in the alphabet.*

2.2 Semantics

The truth value of a formula depends on the meaning assigned to each of the symbols in the formula. The quantifiers and connectives have a fixed meaning but the meaning of constants, function symbols and predicate symbols varies according to the *domain* of discourse. For example, a domain could be a set of families with two children, a set of mechanical structures, or the set of prime numbers.

Each term in the language refers to an object or element in the domain, each function symbol denotes a mapping on the domain and each predicate symbol denotes a relation in the domain. Specifying a meaning for each symbol and determining the truth value of each formula is called an *interpretation* of the language. An interpretation for which a formula expresses a true statement is called a *model* of the formula. We now provide the corresponding definitions.

Definition 2.19 – Pre-interpretation. *A* pre-interpretation *of a first-order language L consists of the following:*

1. *A non-empty set* $\mathbf{D}$*, called the* domain *of the pre-interpretation which may be finite or infinite.*
2. *Each constant in L is assigned an element of* $\mathbf{D}$*.*
3. *Each n-ary function symbol f in L is assigned a mapping from* $\mathbf{D}^n$ *to* $\mathbf{D}$*, where* $\mathbf{D}^n$ *is the set of all n-tuples of domain elements:* $\mathbf{D}^n = \{(d_1, \ldots, d_n) \mid$ *for every* $1 \leq i \leq n, d_i \in \mathbf{D}\}$*.*

Definition 2.20 – Interpretation. *An* interpretation *I of a first-order language L consist of the following:*

1. *A pre-interpretation J, with some domain* $\mathbf{D}$*, of L. I is said to be based on J.*
2. *Each n-ary predicate symbol* $\mathbf{p}$ *in L is assigned a mapping from* $\mathbf{D}^n$ *to* $\{$*true, false* $\}$*.*

Definition 2.21 – Model of a Formula. *Let* ϕ *be a formula, and I be an interpretation of a language L. I is a* model *of* ϕ *if I satisfies* ϕ *(i.e., the truth value of* ϕ *under I is true).*

Definition 2.22 – Model of a Theory. *Let* Σ *be a theory (a set of formulas), and I an interpretation. I is a* model *of* Σ *if I is a model of all formulas* $\phi \in \Sigma$*.*

Definition 2.23 – Logical Consequence. *Let* Σ *be a theory (a set of formulas), and* ϕ *a formula.* ϕ *is a* logical *(or* semantic*)* consequence *of* Σ *(written as* $\Sigma \models \phi$*), if every model of* Σ *is also a model of* ϕ*. Then it is also said that* Σ (logically) implies ϕ*. Similarly, a theory* Γ *is said to be a* logical *(or* semantic*)* consequence *of* Σ *(*$\Sigma \models \Gamma$*), if* $\Sigma \models \phi$*, for every formula* $\phi \in \Gamma$*. Then it is also said that* Σ (logically) implies Γ.

2.3 Programming Aspects

Logical implication for first-order logic (by Church's Theorem [10]) is an undecidable problem. That means that there is no algorithm which can in a finite number of steps find out whether $\Sigma \models \phi$, for every ϕ and Σ. However, there are *proof procedures* which given Σ and ϕ can in a finite number of steps return the right answer if $\Sigma \models \phi$, otherwise the procedures answer 'no' or do not terminate.

Given a set of formulas Σ (the premises) and a set of inference rules $\mathbf{R}$, a *proof procedure* shows the way to derive or construct some formula ϕ (the conclusion). The procedure applies $\mathbf{R}$ to Σ and previously derived formulas until ϕ is derived. The fact that ϕ can be derived from Σ is written as $\Sigma \vdash \phi$. There are two properties which are desirable in a proof procedure: *soundness* and *completeness*. A proof procedure is sound if all formulas it derives from some Σ are logical consequences of Σ; and it is complete if it can derive all logical consequences of Σ.

The two most important inference rules are *resolution* and *subsumption*. Since a formal introduction to resolution is beyond the scope of this chapter, interested readers are referred to [76]. We now define subsumption.

Definition 2.24 – Substitution. *A* substitution θ *is a finite set of the form* $\{X_1/t_1, \ldots, X_n/t_n\}, n \geq 0$, *where the* X_i *are distinct variables and the* t_i *are terms. Then it is said* t_i *is substituted for* X_i. *Each element* X_i/t_i *is called a* binding *for* X_i. *The substitution* θ *is called a* ground substitution *if every* t_i *is a ground term.* θ *is called a* variable-pure substitution *if every* t_i *is a variable.*

Definition 2.25 – Expression. *An* expression *is either a term, a literal, or a conjunction or disjunction of literals.*

Definition 2.26 – Renaming Substitution. *Let* A *be an expression, and let* θ *be the variable pure substitution* $\{X_1/Y_1, \ldots, X_n/Y_n\}$. θ *is a renaming substitution for* A *if each* X_i *occurs in* A, *and* $Y_1, \ldots, Y_n$ *are distinct variables such that each* Y_i *is either equal to some* X_j *in* θ, *or* Y_i *does not occur in* A.

Definition 2.27 – Subsumption. *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be clauses.* $\mathcal{C}$ subsumes $\mathcal{D}$ *if there exists a substitution* θ, *such that* $\mathcal{C}\theta \subseteq \mathcal{D}$ *(i.e., every literal in* $\mathcal{C}\theta$ *is also in* $\mathcal{D}$*).* $\mathcal{C}$ properly subsumes $\mathcal{D}$ *if* C *subsumes* $\mathcal{D}$ *and* $\mathcal{D}$ *does not subsume* $\mathcal{C}$.

Let us illustrate subsumption, assume $\mathcal{C}$ and $\mathcal{D}$ are the following clauses:

$\mathcal{C}$ = **no_payment_due(Person) ← unemployed(Person).**
$\mathcal{D}$ = **no_payment_due(ana) ← unemployed(ana), female(ana).**

by applying the substitution $\theta = \{\text{Person/ana}\}$ to $\mathcal{C}$, $\mathcal{C}$ subsumes $\mathcal{D}$.

If $\mathcal{C}$ subsumes $\mathcal{D}$ then $\mathcal{C}$ logically entails $\mathcal{D}$ ($\mathcal{C} \models \mathcal{D}$); however, the reverse is not always true. That means subsumption is sound but incomplete with respect to logical implication.

From the point of view of ILP, an important property of subsumption is that it introduces a *partial order* on a set of clauses based on their *generality.*

Definition 2.28 – Relation Properties. *Let R be a relation on a set* $\mathbf{S}$.

1. *R is* reflexive *if for all $x \in \mathbf{S}$, xRx holds.*
2. *R is* symmetric *if for all $x, y \in \mathbf{S}$, xRy implies that also yRx.*
3. *R is* transitive *if for all $x, y, z \in \mathbf{S}$, xRy and yRz implies xRz.*
4. *R is* antisymmetric *if for all $x, y \in \mathbf{S}$, xRy and yRx implies $x = y$.*

Definition 2.29 – Partial Order. *A relation R on a set $\mathbf{S}$ is a* partial order *on $\mathbf{S}$, if R is reflexive, transitive, and antisymmetric. A partial order is denoted by $\succeq$.*

Definition 2.30 – Bounds. *Let $\mathbf{S}$ be a set with a partial order $\succeq$ and let $\mathbf{S}' \subset \mathbf{S}$. An element $x \in \mathbf{S}$ is an* upper bound *of $\mathbf{S}'$ if $x \succeq y$ for all $y \in \mathbf{S}'$. An upper bound x of $\mathbf{S}'$ is a* least upper bound (lub) *of $\mathbf{S}'$, if $z \succeq x$ for all upper bounds z of $\mathbf{S}'$. If $x \in \mathbf{S}$ is an upper bound of $\mathbf{S}'$, and if for any upper bound $y \in \mathbf{S}$ of $\mathbf{S}'$ we have that $x \succeq y$ implies $x \approx y$, then x is called a* minimal upper bound (mub) *of $\mathbf{S}'$.*

Similarly, an element $x \in \mathbf{S}$ is a lower bound *of $\mathbf{S}'$ if $y \succeq x$ for all $y \in \mathbf{S}'$. A lower bound x of $\mathbf{S}'$ is a* greatest lower bound (glb) *of $\mathbf{S}'$, if $x \succeq z$ for all lower bounds z of $\mathbf{S}'$. If $x \in \mathbf{S}$ is a lower bound of $\mathbf{S}'$, and if for any lower bound $y \in \mathbf{S}$ of $\mathbf{S}'$ we have that $y \succeq x$ implies $x \approx y$, then x is called a* maximal lower bound (mlb) *of $\mathbf{S}'$.*

Definition 2.31 – Lattice. *A partially ordered set $\mathbf{L}$ is a* lattice *if for every subset $\mathbf{L}'$ of $\mathbf{L}$, a lub of $\mathbf{L}'$ and a glb of $\mathbf{L}'$ exist. $\top$ denotes the* top element *(lub) and $\bot$ denotes the* bottom element *(glb) of $\mathbf{L}$.*

Definition 2.32 – ILP Terminology. *Let T be a set of clauses, $S \subset T$ and $\succeq$ a partial order on T. Then the following definitions are used in ILP:*

- *If $\mathcal{C}, \mathcal{D} \in T$ and $\mathcal{C} \succeq \mathcal{D}$, then $\mathcal{C}$ is called a* generalization *of $\mathcal{D}$ (or $\mathcal{C}$ is* more general *than $\mathcal{D}$), and $\mathcal{D}$ is a* specialization *of $\mathcal{C}$ (or $\mathcal{D}$ is* more specific *than $\mathcal{C}$).*
- *An upper bound $\mathcal{C} \in T$ of S is called a* generalization *of S.*
- *A lub $\mathcal{C} \in T$ of S is called a* least generalization *of S.*
- *A mub $\mathcal{C} \in T$ of S is called a* minimal generalization *of S.*
- *A lower bound $\mathcal{C} \in T$ of S is called a* specialization *of S.*
- *A glb $\mathcal{C} \in T$ of S is called a* greatest specialization *of S.*
- *A mlb $\mathcal{C} \in T$ of S is called a* maximal specialization *of S.*

Subsumption and logical implication are the most commonly used generality orders. Under subsumption, a clause $\mathcal{C}$ is more general than $\mathcal{D}$ ($\mathcal{C} \succ \mathcal{D}$) if $\mathcal{C}$

properly subsumes $\mathcal{D}$. In this case, $\mathcal{D}$ is a *proper specialization* (*refinement*) of $\mathcal{C}$. Generalization is seen as an "upward" step in the partial order, while specialization corresponds to a "downward" step. In Section 3.1.2 we explain how partial orders are used to search for clauses in a hypothesis space.

Note that from a programming point of view, a definite program computes bindings for variables of a definite goal. These bindings correspond to the output of the program. Thus we talk about *input* and *output arguments* of a goal, where the input arguments are bound before the goal is executed. For instance, assume we have a logic program that determines whether an individual must pay back a student loan. In this case, the goal ←**no_payment_due(Person)** can be seen as a request to prove that (∃**Person no_payment_due (Person)**) is a logical consequence of our program; however, it can also be regarded as a request to provide a specific binding for **Person** which makes **no_payment_due(Person)** true in the given interpretation. An *answer* is the output from a program and a goal.

Definition 2.33 – Correct Answer. *Let* Π *be a definite program and* $G = \leftarrow g_1, \ldots, g_k$ *a definite goal, and* θ *be a substitution for variables of* G *(also called an* answer *for* $\Pi \cup \{G\}$*). We say that* θ *is a* correct answer *for* $\Pi \cup \{G\}$ *if* $\Pi \models \forall((g_1, \ldots, g_k)\theta)$*.*

2.4 Summary

Since multirelational learning is concerned with inducing concept definitions generally expressed as logic programs, several terms commonly used in ILP can be traced back to logic programming terminology. This chapter provides an introduction to logic programming in ILP and defines logic programming terminology used throughout the thesis.

3 Learning and Search in Multirelational Learning

This chapter provides an introduction to learning and search in ILP, and together with Chapter 2, defines the terms used throughout the thesis. Comprehensive introductions to ILP can be found in [33, 76, 103]. In Section 3.1, learning issues in ILP such as learning task, covering algorithm and refinement operators are reviewed. In Section 3.2, a brief overview of search terminology and search strategies in multirelational learning is given. Finally, in Section 3.3, some ILP systems are described.

3.1 Learning in ILP

Machine learning deals with the construction of systems capable of the autonomous acquisition and integration of knowledge. Most learning systems acquire and integrate knowledge from experience, i.e., they perform *inductive learning.* Inductive learning can be seen as learning the representation of a function; i.e., given a collection of examples of a function f, return a function h that approximates f [95]. To accomplish this task, learning systems search through a large space of hypotheses to find the hypothesis (generalization) h that best fits the examples given [64].

Machine learning approaches (e.g., neural networks, Bayesian learning, reinforcement learning, genetic algorithms, decision tree learning, multirelational learning, and kernel methods) differ from each other in the representation used for examples, hypotheses and prior knowledge (if available), and whether prior knowledge and feedback are considered in the learning process. However, sometimes machine learning approaches overlap; for example, in [37, 50] Bayesian learning and

ILP are combined, and in [30] reinforcement and multirelational learning intersect. In this section, we describe the task of multirelational learning and how this task can be accomplished.

3.1.1 Learning Task

Most multirelational learning systems represent examples and hypotheses using a Horn language and include in the learning process prior knowledge (called *background knowledge* in ILP terminology). Typically, a multirelational learning system takes as input background knowledge B, which provides the system with information about the domain, and a set of examples E, and has to induce a predicate definition T which defines a relation on the domain (a formal definition of this task is given in Definition 3.3). This predicate definition T is used to classify unseen examples $E^?$, and can be seen as a concept definition or as an approximation of a boolean function whose value is true for all objects or elements belonging to that concept.

To define the ILP learning task, we assume a representation similar to the individual-centered representation [35]. That is, in the examples, terms refer to elements of specific types in the domain, and the target concept or target predicate $\mathbf{p}$ denotes a relation R in the domain among these elements. The training examples E represent a proper subset of the set $\mathbf{X}$ of all elements of those types in the domain for which R holds or not, and it is assumed that E follows a distribution D similar to that of $\mathbf{X}$ over which future test examples $E^?$ are drawn. The training examples can be defined as follows.

Definition 3.1 – Training Examples. *Let R be a relation in a domain, let $\mathbf{X}$ be the set of all possible instances of R in the domain, let $\mathbf{p}$ be a predicate symbol denoting R, and D be a probability distribution over $\mathbf{X}$. Training examples E of $\mathbf{p}$ are a set of ground facts with the same predicate symbol $\mathbf{p}$ (and arity) whose terms are drawn at random according to D from $\mathbf{X}$ such that $E = E^+ \cup E^-$ (positive and negative examples) and $E^+ \cap E^- = \emptyset$. Each example $e \in E$ is classified as a positive or negative instance of $\mathbf{p}$ according to whether R holds or not among the terms in e.*

The learning task definition, which we provide below, fits within the predictive ILP framework and within the *learning from entailment* formalization, which is nowadays the most common ILP problem setting. Note, however, that there are systems (e.g., CLAUDIEN [19], TILDE [6]) which use other concept learning formalizations. Learning formalizations differ from each other in the example representation and the membership function or coverage notion; more details on the relation among various ILP settings can be found in [16]. In addition, for simplicity, we have assumed a *single-predicate* learning problem where all given examples are instances of one predicate **p**; however, *multiple-predicate* learning is also considered by ILP Systems (e.g., CLINT [18] and MOBAL [52]).

Based on these assumptions, we define now classification error and the learning task.

Definition 3.2 – Classification Error and Accuracy. *Let T be a predicate definition of a target concept* **p**, *B be a definite program,* D *be a probability distribution over the set* $\mathbf{X}$ *of all possible instances of* **p**, *and e be an unseen example drawn at random according to* D *from* $\mathbf{X}$. *The* classification error *of T with respect to* **p** *and distribution* D *is the probability $\alpha_T \in [0,1]$ that T will misclassify e. e is misclassified by T if $T \wedge B \models e$ and e is a negative instance of* **p**, *or if $T \wedge B \not\models e$ and e is a positive instance of* **p**. *The* predictive accuracy *of T is $(1 - \alpha_T)$.*

Definition 3.3 – ILP Learning Task. *Let B be a definite program,* **p** *be a target concept,* $\mathbf{X}$ *be the set of all possible instances of* **p**, D *be a probability distribution over* $\mathbf{X}$, *and E be training examples of* **p**. *Then the multirelational learning task is as follows.*

Given: B and E.

Find: A set of Horn clauses T in the hypothesis space (see Definition 3.5 below), such that T is a definition of **p** and T minimizes the expected classification error over future instances $E^?$ drawn at random (the same as E) according to D from $\mathbf{X}$. If an example $e \in E^?$ is classified as a positive instance of **p** by T we also say that e is covered (Definition 3.4) by T.

Definition 3.4 – Covering. *Let $\mathcal{C}$ be a definite program clause, A a ground atom, and B a definite program.* $\mathcal{C}$ covers *A with respect to B if there is a ground substitution θ for $\mathcal{C}$, such that the body of $\mathcal{C}$ is true with respect to B, and the head of $\mathcal{C}$ is equal to A. Let T be a set of clauses, T* covers *A if A is covered by at least one clause $\mathcal{C} \in T$.*

Now we define the hypothesis space where a system searches for T.

Definition 3.5 – Hypothesis Space. *Let B be a definite program, E be training examples of a target concept* **p**, *and* $\mathbf{V}$ *be a non-empty set of variables. The* hypothesis space *consists of all Horn clauses whose head has the predicate symbol* **p** *(and arity) as that of the examples in E and whose literals are constructed with* $\mathbf{V}$ *and the constants, predicates and functions in B and E.*

Due to the large size of the hypothesis space, several ILP systems impose restrictions on the properties of the clauses they can learn, such as *size*, *depth*, and *determinacy* of the clauses, and consider only a subset of the clauses in the hypothesis space. This subset is the *search space* of the system. Other systems (e.g., CLINT [18], FOIL [85], RDT [52]) restrict the choice of first-order language further and use a function-free Horn language as the hypothesis language.

Definition 3.6 – Depth. *Let $\mathcal{C}$ be a clause. The* depth *of $\mathcal{C}$ is the largest variable depth of its variables. The* depth of a variable *in $\mathcal{C}$ is calculated as follows. Consider $\mathcal{C} = \mathbf{p}(X_1, \ldots, X_i, \ldots, X_n) \leftarrow l_1, l_2, \ldots, l_r, \ldots$. All the variables $X_1, \ldots, X_i, \ldots, X_n$ that appear in the head of $\mathcal{C}$ have depth zero, and a variable V appearing for first time in literal l_r has depth $d + 1$ where d is the maximum depth of the variables in l_r that appear in $\mathcal{C}$ before l_r, i.e., they occur in $\mathbf{p}(X_1, \ldots, X_i, \ldots, X_n) \leftarrow l_1, l_2, \ldots, l_{r-1}$.*

For instance, consider the clause in Figure 2.1, the variable *Person* has depth zero because it appears in the head of the clause, and the variables *School* and N have depth one because the only variable which appears in a previous literal is *Person*, which has depth zero. The depth of the clause in Figure 2.1 is equal to the highest depth value of its variables, which is one.

Definition 3.7 – Determinacy. *Let $\mathcal{C}$ be a clause. $\mathcal{C}$ is* determinate *if each of its literals is determinate. A literal l_r is determinate if each of its output variables has exactly one ground binding given the ground bindings of its input variables. A literal can be determinate or non-determinate depending on the relationship between its input and output arguments.*

Let us exemplify determinacy. Consider the predicate **fatherOf**/2, where the first argument is the father and the second argument is the child. The literal **fatherOf**(X, Y) is determinate if Y is the input variable and X is the output variable since the value of variable X is uniquely defined by the value of variable Y (a person Y has only one father X); however, if X is the input variable and Y the output variable, the same literal is non-determinate because the value of Y is not necessarily uniquely defined by the value of X (father X can have more than one child Y).

Definition 3.8 – ij-Determinacy. *Let $\mathcal{C}$ be a clause. If $\mathcal{C}$ is determinate, its variable depth is at most i and the arity of its predicate symbols is at most j, then $\mathcal{C}$ is called ij-*determinate.

3.1.2 Refinement Operators

To search for a clause in the hypothesis space to add to a theory T, most ILP approaches use a *refinement operator*. Given a clause $\mathcal{C}$, a *refinement operator* returns a set of clauses which are either specializations or generalizations of $\mathcal{C}$. Since specialization is seen as a "downward" step in a lattice of clauses, a refinement operator returning a set of specializations of a clause is called a *downward refinement operator*. Analogously, operators which compute generalizations of a clause are called *upward refinement operators*. Refinement operators were introduced by Ehud Shapiro in MIS, which is an implementation of his Model Inference Algorithm restricted to Horn clauses [97].

Definition 3.9 – Refinement Operator. *Let S be a clausal language with a partial order $\succeq$. A* downward refinement operator *for S and $\succeq$ is a function ρ, such that $\rho(\mathcal{C}) \subseteq \{\mathcal{D} \mid \mathcal{C} \succeq \mathcal{D}\}$ for every $\mathcal{C} \in S$.*

An upward refinement operator *for S and $\succeq$ is a function δ, such that $\delta(\mathcal{C}) \subseteq \{\mathcal{D} \mid \mathcal{D} \succeq \mathcal{C}\}$ for every $\mathcal{C} \in S$.*

Refinement operators allow a system to search step-by-step through a generality order of clauses for a correct theory T. This ordering structures the hypothesis space and allows the system to organize the search for clauses to be added to T. Since the ILP learning task has been proven undecidable under logical implication [51], most ILP approaches use subsumption (Definition 2.27) as partial ordering of the hypothesis space (e.g., CLAUDIEN [19], FOIL, ICL [21], Progol [68], TILDE [6]). In fact, most multirelational learning systems use a downward refinement operator for clauses ordered by subsumption.

Usually at the top of the lattice there is a unit clause which covers every example in E and has the same predicate symbol (and arity) as that of the examples. A downward refinement operator ρ normally computes only the set of greatest specializations of a clause $\mathcal{C}$ under subsumption by adding literals to the body of $\mathcal{C}$ and by applying substitutions to $\mathcal{C}$. In fact, many downward refinement operators work by adding to a clause C a literal which is an instance of a predicate available from the background knowledge.

The search space is restricted to clauses that can be obtained by successively applying the refinement operator. This approach works if and only if there is a number of refinement steps to every clause composing at least one adequate theory. The search space induced by a refinement operator can be represented as a *refinement graph*.

Definition 3.10 – Refinement Graph. *Let $\mathcal{C}$ and $\mathcal{D}$ be two clauses. A* refinement graph *Υ (see Figure 3.1) is a directed graph which has the clauses in the search space as nodes and contains an edge from $\mathcal{C}$ to $\mathcal{D}$ if $\mathcal{D}$ is a refinement of $\mathcal{C}$ (i.e., $\mathcal{D} \in \rho(\mathcal{C})$).*

A path in a refinement graph from a clause $\mathcal{C}$ to a clause $\mathcal{E}$ is called a $\rho-$chain. $\rho^d(\mathcal{C})$ is the set of clauses obtained after d-step refinements of some clause $\mathcal{C}$;

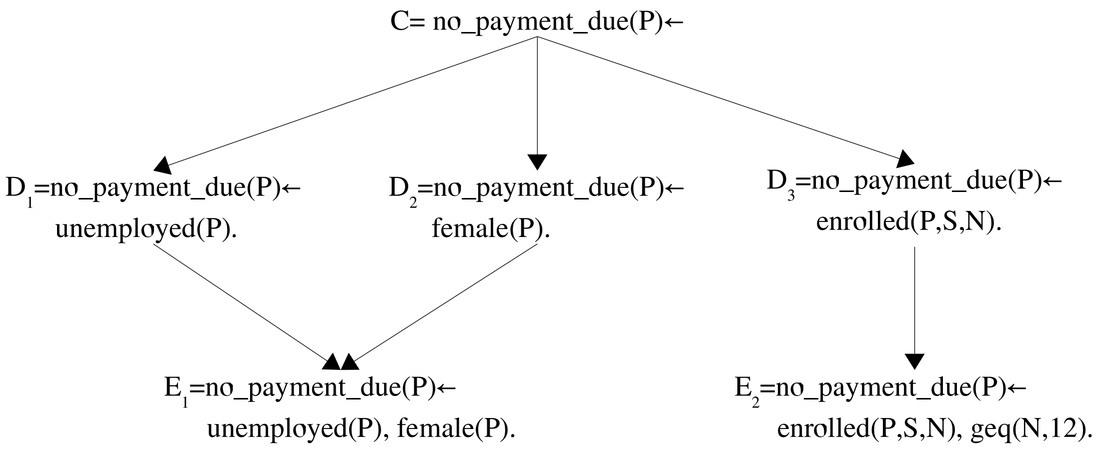

C= no_payment_due(P)←
D1=no_payment_due(P)← unemployed(P).
D2=no_payment_due(P)← female(P).
D3=no_payment_due(P)← enrolled(P,S,N).
E1=no_payment_due(P)← unemployed(P), female(P).
E2=no_payment_due(P)← enrolled(P,S,N), geq(N,12).

Figure 3.1: A partial view of a refinement graph induced by a downward refinement operator under subsumption

for instance, in the refinement graph in Figure 3.1, $\rho^2(\mathcal{C}) = \{\mathcal{E}_1, \mathcal{E}_2\}$. d can be seen as the depth of $\mathcal{E}_1$ and $\mathcal{E}_2$ in the refinement graph. $\rho^*(\mathcal{C})$ is the set of all refinements of $\mathcal{C}$; i.e., $\rho^*(\mathcal{C}) = \rho^1(\mathcal{C}) \cup \rho^2(\mathcal{C}) \cup \ldots$.

For practical purposes, a refinement operator ρ should have certain desirable properties. Ideally, it should be *locally finite*, *complete* and *proper*. If an operator satisfies all these three properties it is called *ideal*. ρ is locally finite if for every clause in the hypothesis space, $\rho(C)$ is finite and computable – otherwise it cannot be used in a system; ρ is complete if there is a finite ρ-chain from a clause to each of its specializations – otherwise ρ might not generate all clauses belonging to a theory; and finally, ρ is proper if it only computes proper specializations of a clause – otherwise the refinement graph might contain cycles and the operator might loop forever. Nienhuys-Cheng and de Wolf [76] proved that *ideal* refinement operators do not exist for clausal languages ordered by subsumption or stronger orders. Thus, since local finiteness is the most important property for practical purposes, refinement operators are either locally finite and proper, or locally finite and complete.

3.1.3 Covering Algorithm

In the previous section, we explain a common approach to search for a single clause. In this section, we introduce the covering algorithm which is the most widespread approach in ILP[1] to learn a set of clauses. Systems using this algorithm build up a theory by sequentially constructing a set of clauses that together cover all or most positive examples and none or few negative examples. Although the goal is to find a theory which minimizes the expected classification error (Definition 3.2), there is not a unique method to decide how well hypotheses will perform on unseen data. Most approaches use heuristics based on statistics about *coverage* and *consistency* of the hypotheses (see [40, 59]) to evaluate the clauses, but other approaches have also been studied (e.g., [99]).

Definition 3.11 – Coverage and Consistency. *Let $\mathcal{C}$ be a clause and E be training examples of a target predicate* $\mathbf{p}$. *The* coverage *of $\mathcal{C}$ with respect to E is the number of examples in E^+ covered by $\mathcal{C}$. The* consistency *of $\mathcal{C}$ with respect to E is the number of examples in E^- covered by $\mathcal{C}$.*

Similarly, the coverage *of a theory T is the number of examples in E^+ covered by T and its* consistency *is the number of examples in E^- it covers. We say T has* complete coverage *with respect to E if $T \models E^+$. T is* consistent *with respect to E if and only if $T \not\models e$, for every $e \in E^-$.*

To illustrate the covering algorithm, assume there is an existing partial theory T_p (which is initially empty), background knowledge B, and training examples E. A system using the covering algorithm constructs first a new clause $\mathcal{C}_{new}$, which covers some positive examples $E^{+\prime}$ and few or none of the negative examples, then removes from E^+ the subset $E^{+\prime}$ of positive examples covered by $\mathcal{C}_{new}$, and adds $\mathcal{C}_{new}$ to the partial theory ($T_p = T_p \cup \mathcal{C}_{new}$). This process is repeated until no positive example remains uncovered ($E^+ = \emptyset$) or no $\mathcal{C}_{new}$ can be constructed. The negative examples determine the level of specialization of every $\mathcal{C}_{new}$ and are therefore kept during the whole process.

The covering algorithm described here learns sets of clauses instead of decision lists (as most multirelational covering systems do). That means that the clauses are unordered, independent, and several of them may apply to a new example which has to be classified. In addition, it is described for single-predicate learning. A way to adapt this algorithm to multiple-predicate (also called multi-class) learning is to separately construct a theory for each predicate. In this case, when clauses for predicate $\mathbf{p}$ are learned, E^+ are the examples belonging to $\mathbf{p}$ and E^- are the examples from all the other classes; this works under the preconditions that all the predicates have the same arity, and that the relations denoted by the other predicates hold for elements of the same types as the relation denoted by $\mathbf{p}$.

Multirelational learning systems using the covering algorithm differ from each other in the approach taken to find $\mathcal{C}_{new}$. In fact, ILP systems can be classified

[1]The reader can consult [40] for a survey of ILP systems using the covering algorithm.

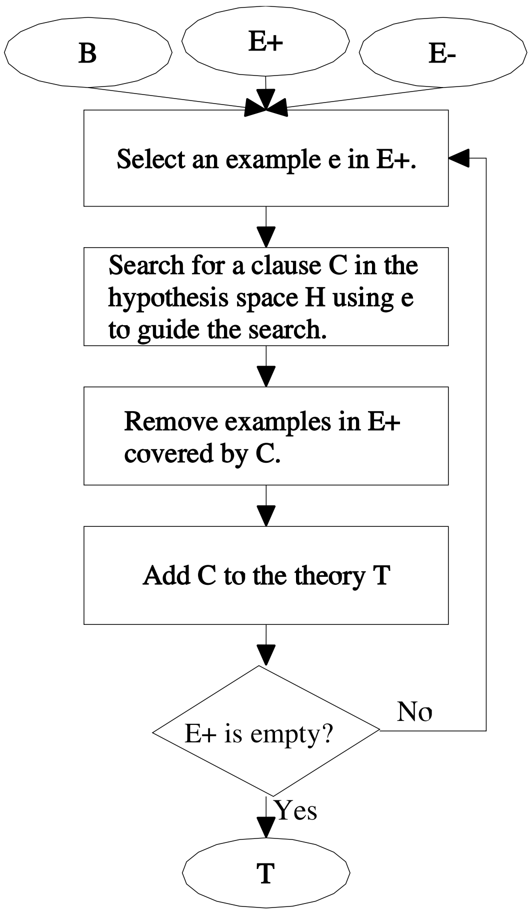

B
E+
E-
Select an example e in E+.
Search for a clause C in the hypothesis space H using e to guide the search.
Remove examples in E+ covered by C.
Add C to the theory T
E+ is empty?
No
Yes
T

Figure 3.2: Example-driven covering algorithm

according to various dimensions. One of these dimensions is the direction, top-down or bottom-up, in which a system searches. In the top-down approach, which is the typical approach taken by ILP systems (e.g., FOIL, ICL, MIS [97], Progol), the search starts with a unit clause and proceeds to consider specializations of this clause. Conversely, in a bottom-up approach the system (e.g., CIGOL [70], GOLEM [72]) searches from an overly specific clause to a more general one.

Another dimension is whether the individual examples are used to constrain the search space of the system (*example-driven* approach), or not (*generate-then-test* approach)[2]. Generate-then-test algorithms (e.g., FOIL, ICL), construct $\mathcal{C}_{new}$ based on heuristic measures that take all examples into account. One positive point of this approach is that, since all examples are taken into account to evaluate a clause, few noisy data does not affect the quality of the learning results;

[2]This terminology is taken from [65].

a negative point is that generate-then-test systems may have trouble learning complex rules. Example-driven algorithms (e.g., CIGOL, GOLEM, Progol), on the other hand, take individual examples as starting points and use them to constrain the search space. An important advantage of the example-driven approach is that individual examples can be used to guide the search for hypotheses. On the negative side, example-driven algorithms are more easily misled by a few noisy examples and are hence less robust when the training data contains errors. Figure 3.2 depicts the example-driven covering algorithm.

Finally, a third dimension is the search strategy used by the system. This strategy can go from exhaustive search in one extreme to hill-climbing search in the other extreme. Search issues in multirelational learning are discussed in Section 3.2.

3.2 Search in Multirelational Learning

Search in ILP covering systems is usually done by traversing the refinement graph (Definition 3.10). In that case, the search algorithm is performed inside the covering algorithm and uses a refinement operator (Definition 3.9) to traverse the graph. Typically, the refinement graph is searched level-wise and the quality of a clause $\mathcal{C}$ is estimated using an heuristic function $eval(\mathcal{C})$ called *evaluation function.*

Definition 3.12 – Evaluation Function. *Let $\mathcal{C}$ be a clause. An* evaluation function *$eval(\mathcal{C})$ returns a real number for $\mathcal{C}$ based on a number of attributes of $\mathcal{C}$. These attributes typically are the coverage, the consistency and the size of $\mathcal{C}$.*

In a top-down approach, the search starts at the top of the lattice, and then, all or a subset of the clauses in the sets $\rho^d(\mathcal{C}), d = 1 \ldots n$ are iteratively evaluated with $eval(\mathcal{C})$ until either a clause to be added to the theory is found, there are no more refinements to consider, or the resources assigned to the search are exhausted. How many clauses are considered at each iteration depends on the search strategy used.

3.2.1 Breadth-first

In breadth-first search, the clause $\mathcal{C}$ at the top of the lattice is considered first, then all the clauses which are refinements of $\mathcal{C}$, then the refinements of $\mathcal{C}$'s refinements, and so on. In general, breadth-first considers all the refinements at depth d in the refinement graph before the refinements at depth $d + 1$. Breadth-first is complete and optimal, but it has exponential complexity.

```
Top_Down_Search
Input: The top of the lattice $\top$, $B$, $E$, $b$
Output: Either a clause $\mathcal{C}$ or $\emptyset$
    1. $\mathbf{S} = \{\top\}$ /* Ordered set of clauses to consider*/
    2. While a clause $\mathcal{D} \in \mathbf{S}$ can be refined or search resources are not
       exhausted
        a) $\mathbf{R} = \emptyset$ /* Set of refinements */
        b) For every clause $\mathcal{D} \in \mathbf{S}$ that can be refined
            i. $\mathbf{R} = \mathbf{R} \cup \rho(\mathcal{D})$
        c) Sort $\mathbf{R}$ according to $eval$
        d) Let $\mathbf{R}_b$ be the best $b$ refinements in $\mathbf{R}$
        e) $\mathbf{S} = \mathbf{R}_b$
    3. Let $\mathcal{C}$ be the best evaluated clause in $\mathbf{S}$
    4. If $eval(\mathcal{C}) \geq$ minimum value accepted and $\mathcal{C}$ covers enough positive
       and few enough negative examples in $E$
        a) Then Return $\mathcal{C}$
        b) Else Return $\emptyset$
```

Algorithm 3.1: Top-down beam-search

With a Horn language as hypothesis language, breadth-first can be applied only to small problem instances because the *branching factor* (i.e., the number of refinements per clause) can be very large depending on the available background knowledge. Typically, ILP systems using breadth-first search use other measures to control the learning complexity (e.g., RDT). Setting $b = \infty$ in step 2d of Algorithm 3.1 yields breadth-first search.

3.2.2 IDA*

Iterative deepening A* (IDA*) [53, 54] explores at each iteration all clauses inside a *heuristic threshold*. This heuristic threshold can for example be the size of the clause, number of literals needed to obtain an *I/O-complete clause* (Definition 4.7) or a minimum evaluation value. If an iteration finishes without finding a solution, the heuristic threshold is extended and the search is restarted. The IDA* algorithm is shown in Algorithm 3.2. IDA* is complete and optimal as long as an *admissible heuristic* is used. Admissible heuristics are optimistic in the sense that they always assume that the goal (in our case, the best refinement of a clause) is closer than it actually is.

```
IDA*
Input: The top of the lattice $\top$, $B$, $E$, threshold $t$
Output: Either a clause $\mathcal{C}$ or $\emptyset$
      1. currentThreshold = $t$
      2. solutionFound = FALSE
      3. Do
         a) S = $\{\top\}$ /* Ordered set of clauses to consider */
         b) Do
            i. Let $\mathcal{C}$ be the first clause in S
            ii. Remove $\mathcal{C}$ from S
            iii. If $eval(\mathcal{C})$ is maximal and $\mathcal{C}$ covers enough positive and few
                 enough negative examples in $E$
               A. Then  solutionFound = TRUE
               B. Else If $\mathcal{C}$ is inside currentThreshold
                    Then S = S $\cup\ \rho(\mathcal{C})$
         c) Until solutionFound or S = $\emptyset$ or search resources exhausted
         d) Extend currentThreshold
      4. Until solutionFound or search resources exhausted
      5. If solutionFound
         a) Then Return $\mathcal{C}$
         b) Else Return $\emptyset$
```

Algorithm 3.2: Top-down IDA* search

3.2.3 Hill-Climbing

Hill-climbing search is one of the most commonly used search algorithms in ILP systems (e.g., FOIL, GOLEM, TILDE) due to its efficiency. Hill-climbing consists in selecting the best refinement of a clause until a clause is reached which cannot be further refined. By setting $b = 1$ in step 2d in Algorithm 3.1, hill-climbing search is performed. If there is more than one best refinement to choose from, hill-climbing randomly selects one or returns the first one found. Contrary to IDA* and breadth-first search, hill-climbing does not guarantee optimality because of the following well-known problems:

1. Local optima (also called myopia). The algorithm does not find a global optimum if any of the refinement steps along the path to get to the global optimum is locally suboptimal.
2. Plateaux. Following the “hill-climbing” metaphor, a *plateau* is an area of the search space where the evaluation function is essentially flat (i.e., it does not differentiate between refinements). The search algorithm performs then a random walk.

3.2.4 Beam-search

Some systems (e.g., m-FOIL [29], ICL) mitigate the problems of hill-climbing search by performing beam-search. Beam-search considers the b best refinements at each level, where b $(1 < b \ll \infty)$ is the beam width. Algorithm 3.1 shows the beam-search algorithm. Although beam-search explores a larger search space than hill-climbing, it does not guarantee optimality either.

3.3 Multirelational Learning Systems

Although Plotkin's seminal work in ILP [84], in which he formalized induction in terms of clausal logic, goes back to 1970, multirelational learning became a flourishing field only until the second half of the 1980s. By the beginning of the 1990s, systems such as FOIL [85], GOLEM [72], LINUS [57] and CIGOL [70] were implemented. With FOIL, Quinlan adapted his attribute-value decision tree algorithm to work using a first-order representation; LINUS solves relational problems by propositionalizing them and applying one of several possible attribute-value algorithms to the propositional representation; CIGOL introduced an approach known as *inverse resolution*, and GOLEM is based on Plotkin's ideas.

In 1990 Muggleton coined the name inductive logic programming, and organized with Pavel Brazdil the first international workshop on ILP. Since then, ILP papers have regularly appeared in major AI and machine learning conferences, and several ILP systems have been applied successfully to real-world problems [8, 28]. ILP Systems are also increasingly being used as tools for data mining in business and marketing applications [105].

There is a vast amount of ILP systems: in the proceedings of the last seven (1997–2003) international conferences on ILP, over sixty different multirelational learning systems have been used for experiments. However, even though there are numerous systems[3], only few of them are frequently used. For example, in the same ILP proceedings, only seven systems are used in more than three papers. In this thesis, we decided to compare our system with the three systems most commonly used in ILP publications (i.e., FOIL [85], Progol [68] and TILDE [6]). Next we briefly describe the seven multirelational learning systems most frequently used in order of relevance.

[3]Overviews can be found in [40, 71, 102].

3.3.1 Progol

Progol [68, 73] (also known as CProgol) is a top-down, example-driven, covering system which performs an A* search through the refinement graph. Progol reduces the hypothesis space by requiring as input, besides a definite program B as background knowledge and a set of ground facts E as examples, a set of *mode declarations*, and by using a *most specific clause* (also called *bottom clause*) as the glb (Definition 2.30) of the refinement lattice. The mode declarations (more about them in Section 4.1) define the predicates from B which can appear in the head and in the body of the clauses in the hypothesis space, as well as the type (and mode) of the arguments taken by the predicates. Mode declarations are a common approach to reduce the branching factor of a refinement graph and have been used since the early days of ILP (e.g., MIS [97]). A bottom clause (Section 4.2) is a maximally specific clause which entails (covers) a positive example e and is derived using *inverse entailment* [68].

Progol has been applied to many application domains and it has become a standard in ILP. However, as will be shown in the next chapter, Progol's search space in structural/topological domains contains unnecessary clauses that should not be considered by the system, and, as will be shown in Chapter 7, its learning results may significantly vary depending on the order in which the system takes the training examples.

3.3.2 TILDE

TILDE [6, 4] is an upgrade to multirelational learning of Quinlan's popular C4.5 algorithm [87] for decision tree induction that uses the *learning from interpretations* [16] setting. In this setting, each example is a Prolog program that encodes the example's description. TILDE, as well as Progol, needs that the user provides (besides B and E) a set of mode declarations, and accepts B in the form of a definite program.

First-order logical decision trees are induced by TILDE in a top-down, generate-then-test fashion using hill-climbing search. It starts with the empty tree, computes all possible refinements that can be applied at that node, evaluates them, and selects the refinement (test) with the highest heuristic value. It uses then the resulting partial tree to classify the examples. All examples passing the test are assigned to one branch of the tree and the rest is assigned to the other branch. This procedure is iteratively applied to each branch until all bottom nodes are leaves. A node can be turned into a leaf if it contains only examples of a single class.

TILDE's main drawback is its myopia when dealing with non-discriminating relations (see Chapter 6). To alleviate this problem, the user can give TILDE a set of templates indicating how to combine those non-discriminating relations with other relations in the domain. Although this approach mitigates TILDE's myopia, it requires extra work for the user.

3.3.3 FOIL

FOIL [85, 89] is a top-down, generate-then-test, covering system that extends to first-order logic methods from attribute-value learning systems. FOIL's language is function-free, and background knowledge and examples have to be extensionally defined as sets of tuples of constants.

FOIL explores the hypothesis space using a slightly modified hill-climbing strategy, where it maintains alternative refinements (*checkpoints*) when the refinement chosen is only marginally better than the alternatives. These alternatives are pursued if the actual choice produces no adequate clause to add to the theory which allows the system to continue searching after hitting a dead end. FOIL's evaluation function is based on the *information gain* heuristic. Several newer ILP systems, e.g., FOCL [78], mFOIL [29] and FFOIL [88], have incorporated elements of FOIL's approach and made extensions on it.

The same as TILDE, FOIL's main drawback is its myopia when dealing with non-discriminating relations (Chapter 6). FOIL's approach to deal with this problem is based on ij-determinate literals (Definition 3.8), however, determinate literals are not suitable for every structural/topological domain because they require determinacy (Definition 3.7).

3.3.4 RIBL

RIBL [34, 46] stands for relational instance-based learning and extends the k-nearest neighbor (kNN) classification approach to a first-order representation. According to [34], this approach is appropriate for domains with continuous attribute values and/or with noisy attributes and/or noisy examples. Background knowledge for this system consists of facts, and, since RIBL delays generalization until classification time, it does not construct an explicit theory.

To our knowledge, experimental results of RIBL on other domains other than Hungarian part-of-speech tagging [45] and mRNA signal structure prediction [46] are not available. For this reason, we lack information about the average performance of RIBL in structural/topological domains.

3.3.5 ALEPH

ALEPH [1] is based on inverse entailment and supersedes P-Progol. Aleph is written in Prolog and, with appropriate settings, can emulate some of the functionality of other ILP systems such as CProgol, FOIL, FORS [48], MIDOS [104], TILDE and WARMR [23].

ALEPH suffers the same problems as Progol; namely, its search space contains unnecessary clauses that should not be considered by the system, ant its learning results may significantly vary depending on the order in which the system takes the training examples.

3.3.6 ICL

ICL [21] upgrades the CN2 algorithm [11] to first-order. It uses the learning from interpretations setting and is a top-down generate-then-test covering system which performs beam-search. To specify the hypothesis language, ICL uses a declarative bias language called DLAB [20]. DLAB templates define the syntax of the clauses that can be learned and are automatically translated in a refinement operator (under subsumption) which defines ICL's search space. Constructing the DLAB templates for ICL implies extra work for the user.

3.3.7 FOIDL

FOIDL [66] is an ILP system for learning first-order decision lists. FOIDL is a descendant of FOIL but employs intensional background knowledge and avoids the need for explicit negative examples.

To our knowledge, FOIDL has been exclusively applied for natural language tasks such as learning rules for producing and analyzing inflectional forms of nouns [31, 62], and inducing rules for forming the past tense of English verbs [66]. Thus, there is not enough information to evaluate FOIDL's performance in structural/topological domains.

3.4 Summary

This chapter discusses learning and search issues in multirelational learning. It defines ILP learning task and describes how this task is achieved using the covering algorithm and refinement operators. In the section dedicated to search in multirelational learning, four search strategies, namely hill-climbing, beam-search, breadth-first and IDA*, are reviewed. Finally, seven frequently-used ILP systems are briefly described to illustrate different multirelational learning implementations.

4 A Basic Approach for ILP

As explained in Section 3.1.3, multirelational learning systems using the covering algorithm differ from each other in various aspects, and usually, new ILP approaches have to be developed considering a set of characteristics common to some systems. For the work presented in this thesis, we decided to focus on the *top-down search* approach. We chose this approach because it is the most common approach in ILP [40], and thus, new developments can be easily adapted to other systems. In addition, since *example-driven learning* significantly improves efficiency over a purely generate-then-test approach [33], we restricted ourselves to the example-driven approach. This chapter describes the particular example-driven top-down approach which serves as base case for this thesis and is henceforth referred to as *literal-based approach*.

To describe the literal-based approach we build upon Chapter 3. Specifically, this approach deals with the learning task defined in Section 3.1.1; it uses a downward refinement operator to search for a clause to add to a theory T (see Section 3.1.2); and it constructs T using the covering algorithm (Section 3.1.3). In the literal based approach, a system is given type and mode information about the arguments of background predicates. This information is usually given to ILP systems in the form of *mode declarations*. The literal-based approach uses mode declarations similar to those used in Progol (Section 3.3.1) and TILDE (Section 3.3.2); these mode declarations are explained in the next section. Section 4.2 describes the use of a bottom clause as a lower bound for the search space, and Section 4.3 defines the literal-based refinement operator.

4.1 Mode Declarations

The *mode declarations* are a set of specifications indicating how to construct literals that can be added to a clause by the refinement operator. The mode declarations contain a predicate symbol and the modes and types of the predicate's arguments. There are two types of mode declarations: those which are used to form a head literal (i.e., a literal having as predicate symbol a target predicate) and are called **modeh**; and those which define body literals and are called **modeb**. Thus, a mode declaration can have one of the following formats:

modeh: target_concept(v_1**type**$_1$**,...,**v_n**type**$_n$**)**
modeb: predicate(v_1**type**$_1$**,...,**v_n**type**$_n$**)**

where v_i and type$_i$ indicate the mode and type of the ith argument, respectively. The mode of an argument is indicated by a +,- or #,[1] and each type$_i$ should be defined in the background knowledge.

In the literal-based approach, each literal used to refine some clause $\mathcal{C}$ is constructed according to a mode declaration. Thus, there is a correspondence between literals and mode declarations.

Definition 4.1 – Correspondence between a literal and a mode declaration. *Let* $m =$ **pred(**v_1**type**$_1$**,...,**v_i**type**$_i$**,...,**v_n**type**$_n$**)** *be a mode declaration, and* $l =$ **pred(**$X_1, \ldots, X_i, \ldots, X_n$**)** *be a literal constructed according to* m*. Then we say that* v_i**type**$_i$ *corresponds to* X_i *so that* X_i *is said to be an* input *(respectively* output*)* variable *of type* **type**$_i$ *when* v_i *is + (-). In this case, we also say that* l has *an input (output) variable of type* **type**$_i$*. If* v_i *is #, a constant* c *is substituted for* X_i *and we say that* c *is a constant of type* **type**$_i$ *and* l has *a constant of type* **type**$_i$*.*

A literal l is constructed according to a mode declaration m using Algorithm 4.1.

For example, the following mode declarations can be used to learn the target concept **auntOf(A,B)**, which means "aunt of **A** is **B**".

modeh: auntOf(+person,+person)
modeb: parentOf(+person,-person) /* parent of **+person** is **-person** */
modeb: sisterOf(+person,+person)
modeb: brotherOf(+person,+person)

In this case, the head literal of every clause considered has the target predicate **auntOf** and two input variables of type **person**, and each body literal of every clause constructed has as predicate symbol **parentOf**, **sisterOf** or **brotherOf**.

[1]In Section 5.1, we extend the mode declaration language.

```
Input: a mode declaration $m$ = pred($v_1$type$_1$,...,$v_i$type$_i$,...,$v_n$type$_n$), some
    clause $\mathcal{C}$ to refine, a non-empty set of variables V, $e \in E^+$ and $B$.
Output: literal $l$
    1. Let $l$ have predicate symbol pred
    2. For every $v_i$type$_i$ in $m$
       a) If $v_i$ is + then
             Add to $l$ a variable $X_i \in$ V such that $X_i$ occurs in the head
                literal of $\mathcal{C}$ as input variable of type type$_i$ or a body literal in
                $\mathcal{C}$ has $X_i$ as output variable of type type$_i$.
       b) Else If $v_i$ is - then
             Add to $l$ a variable $X_i \in$ V such that $X_i$ may appear only in
                the head literal of $\mathcal{C}$ as output variable of type type$_i$.
       c) Else Add to $l$ a constant $c$ of type type$_i$ obtained from $e$ and $B$.
```

Algorithm 4.1: Constructing a literal according to a mode declaration

We classify the literals according to their variables as *providers* or *consumers.* That is, a literal is a *provider* if it has an output variable ($-var$) and a *consumer* if it has an input variable ($+var$). We also establish provider-consumer relations among the literals: A literal l is a *consumer* of literal q if l has at least one $+var$ bound to an $-var$ of q; conversely, q is a *provider* of l. Notice that these relations apply as well to the head literal. A literal providing the value for an output variable of the head literal is a *head provider*, and a body literal is a *head consumer* if it consumes a $+var$ of the head.

We now precisely define the hypothesis space defined by a set of mode declarations.

Definition 4.2 – Hypothesis Space Defined by a Set of Mode Declarations. *Let mh be a mode declaration specifying a target predicate* **p**, *M be a set of* **modeb** *declarations, E be training examples of* **p** *(Definition 3.1), B be a definite program containing a predicate definition for each predicate specified by an $mb \in M$, and* **V** *be a non-empty set of variables.*

The hypothesis space *H^m defined by mh and M consists of all Horn clauses whose head has the predicate symbol* **p** *(and arity) as defined by mh, and whose body literals are constructed according to an $mb \in M$ using Algorithm 4.1.*

Definition 4.2 determines the hypothesis space with respect to a single head declaration *mh*. If several head declarations are given, the complete hypothesis space is the union of all hypothesis spaces defined per **modeh**.

By constructing the literals according to the types (and modes) indicated by the mode declarations, we enforce *type strictness.*

```
modeh: water_quality(+river,#class).
modeb: warmer_than(+temperature,#temperature).
modeb: water_chlorine_concentration(+river, -chlorine_concentration).
⋮
% Type definitions
temperature(X):- float(X).
chlorine_concentration(X):- float(X).
```

Table 4.1: Sample type definitions and mode declarations to illustrate type strictness

Definition 4.3 – Type Strictness. Type strictness *is achieved when*

(a) *every variable is assigned a unique type (in our case this type assignment is done through the correspondence between literals and mode declarations),*
(b) *a variable may be only bound to a term of the variable's type, and,*
(c) *in a correct answer (see Definition 2.33), each term substituted for a variable is assigned the type of the variable.*

Let us illustrate this concept. Suppose the task is to learn a theory about river water quality and some background predicates deal with water temperature and others with chlorine concentration (see Table 4.1). The user may decide then to define a type called **temperature** and another one **chlorine_concentration** as shown in Table 4.1. In this case, we say that **temperature** and **chlorine_concentration** are *compatible types*; i.e., they map to the same set of values. However, under type strictness, a variable of type **temperature** is not instantiated with a **chlorine_concentration** value (although both types are represented by floats), and thus the following clause is not in H^m.

$$\textbf{water\_quality(X,good)} \leftarrow \textbf{water\_chlorine\_concentration(X,Y),}\\ \textbf{warmer\_than(Y,15.30).}$$

Some ILP systems such as FOIL and Progol require type definitions as well but do not enforce type strictness; i.e., they allow the instantiation of variables with values of compatible types and consider clauses such as the one above (which is semantically incorrect). Thus, to discard meaningless clauses from the hypothesis space H^m, we assume type strictness. Furthermore, type strictness has also advantages for the users because they do not need to introduce artifices in the background knowledge, such as employing different sets of values, to distinguish between types. Type strictness can also be obtained by using a strongly typed language instead of Prolog as proposed by Flach et al. [36]; however, this implies introducing a new language to a field where Prolog is already a standard.

4.2 Bottom Clause

A *bottom clause* or *most specific clause* (see Definition 4.4, below) is constructed with two purposes: 1) to provide a lower bound for the search space, i.e., no learned clause is more specific than the bottom clause; and 2) to guide the search. The latter is done by taking the literals in the bottom clause as the set of literals available to refine a clause (as will be explained in Section 4.3). A most specific clause $\bot$ acts as the bottom element of a refinement lattice (see Definition 2.31).

A bottom clause can be constructed as the relative generalization of two (or more) examples [72], or as the most specific inverse resolvent of an example [70], or with inverse entailment [68]. We derive $\bot$ using inverse entailment because, by using inverse entailment, 1) any clause which could be added to an appropriate theory subsumes the bottom clause, and 2) intensional background knowledge can be used. Yamamoto [107, 106] has pointed out that inverse entailment is complete with respect to Plotkin's relative subsumption if E^+ is not a tautology and $B \not\models E^+$, but it is incomplete with respect to logical implication. This means that there are correct[2] hypotheses which cannot be found since there are B and E^+ where the equivalence $B \wedge Hyp \models E^+ \longleftrightarrow B \wedge Hyp \models \bot_\infty$ does not hold for a correct hypothesis Hyp. Furukawa et al. [42] and Muggleton [69] have proposed conditions under which inverse entailment should be complete for logical implication.

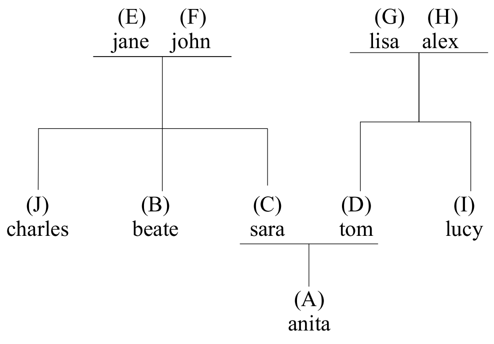
To construct the bottom clause by inverse entailment, one takes a positive example $e \in E$, background knowledge B, and a set of mode declarations. An algorithm to construct $\bot$ can be found in Table 7.6 of [73]. Assuming that literals in a clause are evaluated from left to right, a literal l can appear in $\bot$ if and only if for each $+var$ of l there is at least one provider in $\bot$ and this provider is placed to the left of l in $\bot$. Thus, the literals in $\bot$ are placed according to their consumer-provider relationships. The most specific clause is then defined as follows (cf. Muggleton [68]).

Definition 4.4 – Most Specific Clause $\bot$. *Let $\bot$ be the most specific definite clause constructed with the literals defined by a set of mode declarations, background knowledge B, and example $e \in E^+$ such that:*

(a) $B \wedge \bot \vdash_k e$ *(i.e., e can be derived in k resolution steps), and*

(b) $\bot \succeq \bot_\infty$ *(i.e., $\bot$ subsumes $\bot_\infty$) where $\bot_\infty$ is the (potentially infinite) conjunction of ground literals which are true in all models of $B \wedge \neg E$ ($B \wedge \neg E \models \bot_\infty$).*

[2]According to Yamamoto a hypothesis Hyp is correct if $B \wedge Hyp \models E^+$ and $B \wedge Hyp$ is consistent.



```
    person(anita).           sisterOf(beate,sara).
    person(beate).           sisterOf(sara,beate).
          ⋮                  sisterOf(tom,lucy).
     person(tom).

parentOf(anita,sara).   brotherOf(beate,charles).
parentOf(anita,tom).     brotherOf(sara,charles).
          ⋮                brotherOf(lucy, tom).

 parentOf(tom,alex).       % Positive examples
 parentOf(tom,lisa).        auntOf(anita,lucy).
                            auntOf(anita,beate).
```

Figure 4.1: Kinship example

By using the bottom clause, the hypothesis space H^m (Definition 4.2) is further restricted to $H^\perp$.

Definition 4.5 – Hypothesis Space Defined by a Set of Mode Declarations and a Bottom Clause. *Let mh,* **p**, *M, E, and B be as defined in Definition 4.2; $\perp$ be the bottom clause constructed with mh, M, B and $e \in E^+$; $\top$ be the unit clause whose head has the predicate symbol* **p** *(and arity) as defined by mh; and, H^m be the hypothesis space defined by mh and M according to Definition 4.2.*

The hypothesis space $H^\perp$ *defined by mh, M and $\perp$ consists of every Horn clause $\mathcal{C} \in H^m$ such that $\top \succeq \mathcal{C} \succeq \perp$. We say then that $\mathcal{C}$ is* between $\top$ *and* $\perp$.

From now on, we refer to any bottom clause constructed given a set of mode declarations M, a definite program B and an example $e \in E^+$ simply as $\perp$. In addition, we consider $\perp$ as an ordered sequence of literals where every literal is mapped to its position in $\perp$ starting with the first body literal after the head of $\perp$ (i.e., $p(X) \leftarrow l_1, \ldots, l_{i-1}, l_i, \ldots, l_n$). The literals in $\perp$ are then ordered according to this mapping. We refer henceforth to every literal l_i $(1 \leq i \leq n)$ in $\perp$ as the integer i corresponding to its position in $\perp$.

To illustrate the most specific clause assume we want to learn a theory about the relation **auntOf**. To learn this, we have the genealogical tree depicted in Figure 4.1 on the left, the table shown in Figure 4.1 right, and the positive example: e =**auntOf(anita,beate)** (aunt of anita is beate). The following bottom clause $\perp$ is constructed with e, the background knowledge in Figure 4.1, and the mode declarations listed on page 34. In Figure 4.1, the letters above the names in the genealogical tree indicate the bindings of the variables in the bottom clause.

auntOf(A,B)← -1- parentOf(A,C), -2- parentOf(A,D), -3- parentOf(B,E), -4- parentOf(B,F), -5- parentOf(C,E), -6- parentOf(C,F), -7- parentOf(D,G), -8- parentOf(D,H), -9- sisterOf(B,C), -10- sisterOf(C,B).

Remember that the numbers which precede the literals in the bottom clause indicate the position of each literal in $\perp$ and are used from now on to denote the literals in $\perp$.

4.3 Literal-Based Refinement Operator

The literal-based refinement operator (henceforth referred to as ρ_L) consists in adding one literal from $\perp$ to a clause $\mathcal{C}$ and is defined as follows.

Definition 4.6 – Literal-Based Refinement Operator ρ_L. *Let i and j be literals in $\perp$, and $\mathcal{C}$ be a clause whose last literal is i, then $\rho_L(\mathcal{C}) = \{\mathcal{C} \cup \{j\} \mid i < j$, at least one provider for every $+var$ in j is already in $\mathcal{C}$ and there is at least one provider for every $-var$ in the head of $\mathcal{C}$ in $\mathcal{C} \cup \{j\}\}$.*

The first restriction in the refinement operator achieves non-redundancy in the generation of clauses and the second and third restrictions ensure *I/O-completeness.* In addition, ρ_L preserves the order of the literals in $\perp$.

Definition 4.7 – I/O-completeness. *Let $\mathcal{C}$ be a clause and assume the literals in $\mathcal{C}$ are evaluated from left to right. $\mathcal{C}$ is an* I/O-complete *clause if $\mathcal{C}$ has neither uninstantiated (unbound) output arguments in the head nor uninstantiated input arguments in the body.*

As explained in Section 3.1.2, ideal refinement operators for subsumption do not exist; i.e., a refinement operator has to be either locally finite and proper, or locally finite and complete. In this case, ρ_L is finite and proper but incomplete with respect to the hypothesis space $H^{\perp}$ (Definition 4.5) because the body literals in the constructed clauses maintain the positional order of $\perp$, which is done to achieve ρ_L properness (and I/O-completeness in the clauses).

ρ_L incompleteness with respect to $H^{\perp}$ have two consequences. First, ρ_L generates the clause $\mathcal{D}_1 = \mathbf{p(X)} \leftarrow \mathbf{i}, \mathbf{j}$ but it does not generate $\mathcal{D}_2 = \mathbf{p(X)} \leftarrow \mathbf{j}, \mathbf{i}$; although $\mathcal{D}_2$ may constitute a valid clause. Theoretically, this is not a problem because clauses consist of a disjunction of literals and disjunction is commutative. However, from a procedural point of view, if computations are involved, clauses with the same but permuted literals may not have identical meaning. Second,

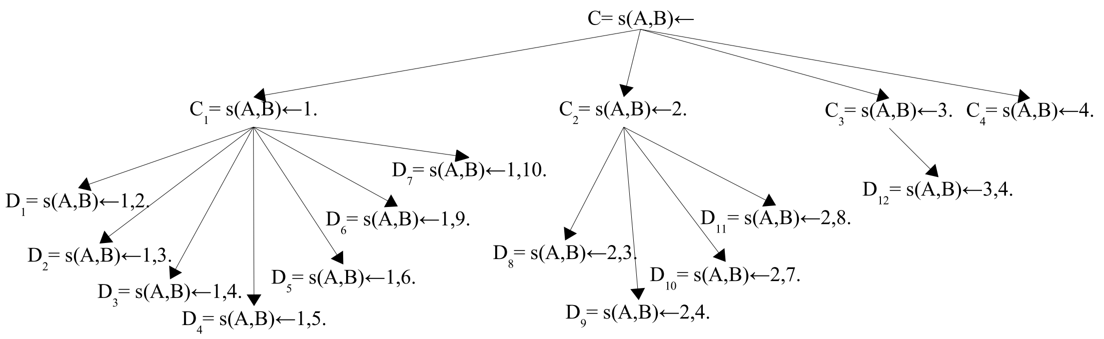


Figure 4.2: Partial refinement graph of ρ_L for the kinship example

suppose we have two literals j and k, and k is a provider of j situated to the right of j in $\perp$ ($j < k$). In this case, ρ_L does not generate either $\mathcal{D}_3$ =**p(X)** ← **j, k** (because when ρ_L considers j there is no provider of j already in $\mathcal{D}_3$) or $\mathcal{D}_4$ =**p(X)** ← **k, j** (because once k is in $\mathcal{D}_4$, j is not added by ρ_L to $\mathcal{D}_4$).

Remark 4.1. The *search space S* explored by ρ_L consists of every clause $\mathcal{C} \in H^{\perp}$ (Definition 4.5) such that $\mathcal{C}$ is I/O-complete (Definition 4.7) and its body literals occur in the same order as in $\perp$.

Let us illustrate ρ_L using again the kinship example for which we constructed the bottom clause in Section 4.2. In this example, every clause explored by ρ_L is between **auntOf(A,B)←** and $\perp$. Figure 4.2 shows the refinement graph (Definition 3.10) of ρ_L for clauses with up to two literals. In Figure 4.2, each body literal is represented by the integer corresponding to its position in $\perp$, and the target predicate **auntOf(A,B)** is represented by **s(A,B)**. By using ρ_L, a multirelational learning system searches in this refinement graph for clauses to add to an appropriate theory about **auntOf(A,B)**. We call those clauses added to the final theory *solutions*.

Definition 4.8 – Solution. *Let Υ be a refinement graph whose top element is $\mathcal{C}$ and whose bottom element is $\perp$ such that $\perp \in \rho^n(\mathcal{C})$; let m be an integer number in $[0, n]$, and let eval be an evaluation function (Definition 3.12). A* solution *is a clause $\mathcal{E} \in \rho^m(\mathcal{C})$ such that there is no $\mathcal{E}' \in \rho^*$ such that eval$(\mathcal{E}')$ > eval$(\mathcal{E})$ or such that eval$(\mathcal{E}')$ = eval$(\mathcal{E})$ and the ρ-chain from $\mathcal{C}$ to $\mathcal{E}'$ is shorter than the ρ-chain from $\mathcal{C}$ to $\mathcal{E}$.*

In the kinship example, clause $\mathcal{D}_7$ corresponds to the following solution.

auntOf(A,B) ← parentOf(A,C), sisterOf(C,B).

4.4 Summary

In this chapter we precisely describe the literal-based approach which is the base case for the work presented in the next chapters. For that, mode declarations, bottom clause and a literal-based refinement operator are defined and illustrated.

5 Macro-operators

In this chapter, we present *macro-operators* (or *macros* for short). Macro-operators are a formal approach to significantly reduce the search space[1] explored in structural/topological domains, which does not further restrict the underlying hypothesis space; i.e.,macros do not affect the completeness of the search. This chapter extends the kinship example (Chapter 4) and relies primarily on the terminology and definitions introduced in Section 3.1.2, Section 3.2 and Chapter 4.

Macros are inspired by and obtained their name from the concept of macro-moves found in the literature about single-agent search in games and puzzles [47, 55]. Korf [55] introduced macros as a domain-independent weak-method for learning and defined a macro as a sequence of operators chosen from the "primitive operators" of a problem. We adjust this definition to multirelational learning by considering the primitive operators to be the literals which form the clauses in the hypothesis space, so a macro is informally speaking a sequence of those literals.

As we will explain in Section 5.1, macro-operators exploit the fact that, in many cases, literals which introduce existential variables do not have discriminative power (i.e., they do not help differentiate between positive and negative examples), and thus, do not need to be included in clauses by themselves; for example, the literal **multiply(A,B,C)** does not have discriminative power because there is always an output value **C** for every combination of input values **A** and **B**. We call such literals *dependent providers* and a list of them (user declared or automatically determined) are the basis for automatically creating the set of macro-operators.

[1]Recall from Section 3.1.1 that in this thesis the search space is the hypothesis space actually searched by a learning system.

In Section 5.2, after formally defining macro-operators, we modify a refinement operator so that it computes the refinements of a clause based on a set of automatically created macro-operators. By using macro-operators instead of single literals, the refinement operator only generates relevant clauses. At the same time, macros only discard clauses that could be pruned from any final theory because they contain unnecessary literals. Section 5.3 provides complete and correct algorithms for constructing macros. Finally Section 5.4 contains the results of the empirical evaluation, and Section 5.5 discusses macros' related work.

5.1 Reducing the Search Space

Sometimes there are literals which, when added to any clause $\mathcal{C}$ in the hypothesis space, do not affect the coverage nor the consistency of it; i.e., they do not help to discriminate among the examples. These literals introduce existential variables in $\mathcal{C}$ (they provide a value). For example, returning to the kinship example of Chapter 4 (see pages 34 and 38), the literal **parentOf(A,F)** is (with respect to the given background knowledge, training examples, and mode declarations) true for every input argument A. Intuitively this is explained by the fact that in the real life everyone has parents. Adding this literal by itself to any clause $\mathcal{C}$ in the hypothesis space of the kinship example does not help to distinguish between the positive and the negative examples of the target concept **auntOf(A,B)**; however, it is necessary to construct any solution. We call these literals *dependent providers.*

Definition 5.1 – Dependent Provider. *Let i be a literal in the body of $\perp$,*[2] *B be a definite program, E be training examples, m a mode declaration defining i,* **VI** *the set of input variables of i according to m,* **VO** *the set of output variables of i according to m, and $\mathcal{C}$ be any clause in $H^{\perp}$ which does not contain i and where all the variables in* **VI** *appear.*

i is a dependent provider *if* **VO** *is non-empty and each variable in* **VO** *has, for the given $B \cup E$, at least one ground binding given the ground bindings of the variables in* **VI**.

In other words, a literal i is a dependent provider if and only if i has output variables and is true for every combination of input argument values.

The literal-based refinement operator defined in Section 4.3 produces clauses $\mathcal{D} \in \rho_L(\mathcal{C})$ which might vary from $\mathcal{C}$ only in having one dependent provider

[2]Remember that we refer to any bottom clause constructed given a set of mode declarations M, a definite program B and an example $e \in E^+$ as $\perp$.

added. However, since dependent providers succeed for every combination of input argument values, they modify neither the coverage nor the consistency of $\mathcal{C}$ but they do increase its length (therefore, $eval(\mathcal{C}) > eval(\mathcal{D})$). In this case, $\mathcal{D}$ cannot be a solution and can be discarded.

Theorem 5.1. *Let $\mathcal{D}$ and $\mathcal{C}$ be clauses such that $\mathcal{D} \in \rho(\mathcal{C})$ and i be a dependent provider in $\mathcal{D}$. $\mathcal{D}$ is not a solution if there is not at least one consumer of i in $\mathcal{D}$ too.*

Proof by contradiction. Assume that $\mathcal{D} = \mathcal{C} \cup \{i\}$ is a solution and i is a dependent provider and there is not a consumer of i in $\mathcal{D}$. But, since i does not modify the statistics about $\mathcal{D}$ except for $\mathcal{D}$'s size, then $eval(\mathcal{C}) > eval(\mathcal{D})$. Hence $\mathcal{D}$ cannot be a solution (Definition 4.8) which contradicts the assumption and completes the proof. □

One way to avoid that $\mathcal{D}$ differs from $\mathcal{C}$ only in having one dependent provider added is 1) to identify the dependent providers in the mode declarations, and 2) to modify the refinement operator so that it only adds dependent providers to $\mathcal{C}$ together with at least one of their consumers. Such a sequence of literals composed of a dependent provider and at least one of its consumers is what we call a *macro-operator*, which is formally defined in the next section.

To identify the dependent providers, we enhance the mode declaration language defined in Section 4.1 with the $*var$ notation. A $*var$ is an output argument of a dependent provider. For instance, with the new notation the mode declaration for the literal **parentOf(A,F)** is **parentOf(+person,*person)**, which means that literals with the predicate symbol **parentOf** are dependent providers which supply an output argument value of type **person**. Using this notation, the types of the arguments in the mode declarations can now be preceded by a +,-,* or #, where - and * are mutually exclusive. That is, in a mode declaration can appear either - or * but not both.

Now we have to modify ρ_L to restrict, based on Theorem 5.1, the search space S (Remark 4.1 on page 40) to:

$S' := \{\mathcal{C} \in S$ such that each dependent provider in $\mathcal{C}$ has at least one consumer for one of its $*var$ in $\mathcal{C}\}$.

We say that the clauses belonging to S' are admissible. In the next section we propose a refinement operator which generates only admissible clauses.

5.2 Macro-based Method

So far we have intuitively used the term macro in relation to a refinement operator, but now we are ready to formally define it. For that, we first define what a legal subsequence of literals is. Remember that we represent the literals as the index corresponding to their positions in the bottom clause. Thus, if i and j are two literals and $i < j$ then literal i appears in $\perp$ to the left of literal j. Note that although the bottom clause is constructed by inverse entailment, this is not a requirement of the macro-based method and thus it is not assumed in the following definitions.

Definition 5.2 – Subsequence of Literals. *Let the body of* $\perp$ *be a sequence of literals* $1, \ldots, n$ *and* $j_k \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, i\}$ *then* $j_1 \ldots j_i$ $(i \leq n)$ *is a* subsequence of literals *when* $j_k < j_{k+1}, \forall k \in \{1, \ldots, i-1\}$.

Observe that the order of the literals in $\perp$ is preserved in a subsequence of literals.

Definition 5.3 – Legal Subsequence of Literals. *A subsequence of literals* $j_1 \ldots j_i$ *is a* legal subsequence *if and only if:*

(a) *at least one* $*var$ *of every dependent provider* j_p, $1 \leq p \leq i$, *is used by a consumer* q *such that* $q \in \{j_{p+1}, \ldots, j_i\}$, *or* j_p *is also a head provider; and,*

(b) *every* $+var$ *of consumer* j_s, $1 < s \leq i$, *has at least one provider* r *such that* $r \in \{j_1, \ldots, j_{s-1}\}$, *or the* $+var$ *is provided by the head.*

$\emptyset$ *is also a legal subsequence of literals.*

Definition 5.4 – Legal Subsequence of Literals Given Another Subsequence. *Let* $j_1 \ldots j_i$ *be a legal subsequence of literals according to Definition 5.3. Then* $j_{i+1} \ldots j_m$ $(m \leq n)$ *is a* legal subsequence of literals given $j_1 \ldots j_i$ *if and only if:*

(a) *at least one* $*var$ *of every dependent provider* j_p, $i+1 \leq p \leq m$, *is used by a consumer* q *such that* $q \in \{j_{p+1}, \ldots, j_m\}$, *or* j_p *is also a head provider; and,*

(b) *every* $+var$ *of consumer* j_s, $i+1 < s \leq m$, *has at least one provider* r *such that* $r \in \{j_1, \ldots, j_{s-1}\}$, *or the* $+var$ *is provided by the head.*

In the previous definitions, the first condition is the restriction we propose for constructing admissible clauses and the second one refers to I/O-completeness; i.e., it ensures that no unbound input variable occurs in a body literal. Definition 5.3 defines subsequences of literals which are legal without requiring the presence of any other body literal in any clause $\mathcal{C}$ to be refined (i.e., they can be added to a unit clause consisting only of the head of $\perp$); while Definition 5.4 refers to subsequences of literals which require that other body literals appear in $\mathcal{C}$ before they can be added to $\mathcal{C}$.

Definition 5.5 – Macro-operator. *A* macro-operator *(or macro for short) is a subsequence of literals $j_{i+1} \dots j_m$ which is either legal or for which there exists a subsequence of literals $j_1 \dots j_i$ ($j_i < j_{i+1}$) such that $j_{i+1} \dots j_m$ is legal given $j_1 \dots j_i$; and no proper subsequence (besides $\emptyset$) of $j_{i+1} \dots j_m$ is legal or is legal given any $j_1 \dots j_i$.*

5.2.1 Macros' Ordering

In the literal-based refinement operator (ρ_L), the literals are ordered by their position in $\perp$ and this ordering is used to achieve non-redundancy in the generation of clauses. With the same purpose, we introduce for the macro-based method a new ordering based on the *maximum provider* of the literals in a macro.

Definition 5.6 – Maximum Provider of a Literal. *Let j be a consumer. The* maximum provider *of j ($max\_provider\_of\_literal(j)$) is the provider i of j ($i < j$) with the greatest (rightmost) position in $\perp$. The position of the head of $\perp$ is 0.*

Definition 5.7 – Maximum Provider of a Subsequence of Literals. *The* maximum provider of a subsequence of literals *$a = j_i \dots j_m$ is:*

$$max\_prov(a) = \max_{k \in \{i \dots m\}} (max\_provider\_of\_literal(j_k)).$$

Definition 5.8 – Comparison between Subsequences of Literals. *Let $a = j_i \dots j_m$ and $b = j_k \dots j_n$ be subsequences of literals, then $a < b$ if and only if:*

(a) *$max\_prov(a) < max\_prov(b)$; or,*
(b) *$max\_prov(a) = max\_prov(b)$ and a is lexicographically less than b. That is, it exists $g \in \{0, \dots, min(m-i, n-k)\}$ such that $j_i = j_k \wedge j_{i+1} = j_{k+1} \wedge \dots \wedge j_{i+g-1} = j_{k+g-1} \wedge j_{i+g} < j_{k+g}$.*

For instance, suppose we obtain the following two macros in the kinship example: $\mathbf{m}_a$ = **parentOf(A,C), sisterOf(B,C)** and $\mathbf{m}_b$ = **parentOf(A,C), sisterOf(C,B)**, which are formed by the literals [1,9] and [1,10], respectively. The maximum provider of both macros is literal 1, since the maximum provider of literal 1 is 0 and the maximum provider of literals 9 and 10 is 1 (i.e., $max\_prov(\mathbf{m}_a) = max\_prov(\mathbf{m}_b)$). Then, according to the second item of Definition 5.8, $\mathbf{m}_a < \mathbf{m}_b$ because [1,9] is lexicographically less than [1,10].

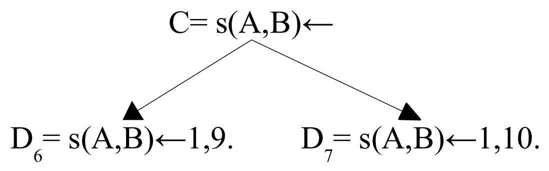


Figure 5.1: Refinement graph of ρ_M for the kinship example

5.2.2 Macro-based Refinement Operator

Now everything is ready to define the macro-based refinement operator (henceforth referred to as ρ_M). Note that single literals can fulfill the macro's definition and be used in the refinement operator defined below.

Definition 5.9 – Macro-based Refinement Operator ρ_M. *Let* $\mathbf{m}_a$ *and* $\mathbf{m}_b$ *be macros and* $\mathcal{C}$ *be a clause (and a legal subsequence of literals) whose last added macro is* $\mathbf{m}_a$*, then* $\rho_M(\mathcal{C}) = \{\mathcal{C} \cup \{\mathbf{m}_b\} \mid \mathbf{m}_a < \mathbf{m}_b$*,* $\mathbf{m}_b$ *is a legal subsequence given* $\mathcal{C}$*, and* $\mathcal{C} \cup \{\mathbf{m}_b\}$ *is I/O-complete*$\}$*.*

Using the macro-based ρ_M the search space S is reduced to legal subsequences of literals.

Remark 5.1. The *search space* S' explored by ρ_M consists of every clause $\mathcal{C} \in H^{\perp}$ such that $\mathcal{C}$ is I/O-complete and a legal subsequence of literals. That is, S' consists of every clause $\mathcal{C} \in S$ such that $\mathcal{C}$ is a legal subsequence of literals.

Let us illustrate the search space explored by ρ_M using our long-familiar kinship example. The bottom clause is the same as the one on page 38, and the only change in the mode declarations is the definition of the predicate **parentOf** from **parentOf(+person,-person)** to **parentOf(+person,*person)**. The new mode declarations are as follows.

modeh: auntOf(+person,+person)
modeb: parentOf(+person,*person)
modeb: sisterOf(+person,+person)
modeb: brotherOf(+person,+person)

In this example, the macros $\mathbf{m}_a$ = **parentOf(A,C), sisterOf(B,C)** and $\mathbf{m}_b$ = **parentOf(A,C), sisterOf(C,B)**, formed by the literals [1,9] and [1,10], are obtained using the algorithms described in Section 5.3. Figure 5.1 shows the refinement graph induced by the macro-based refinement operator (see the refinement graph in Figure 4.2 on page 40 as comparison with ρ_L). ρ_M generates only two clauses while ρ_L generates sixteen clauses.

We now prove the crucial property of the macro-based approach, namely that ρ_M is as complete as ρ_L. For this proof, we introduce the concept of dependent consumer. A *dependent consumer* is a literal which has at least one input argument value provided exclusively by dependent providers. By the macro's definition, a dependent consumer appears always in a macro together with at least one dependent provider.

Theorem 5.2. *Let $\mathcal{D}$ be a solution and $\mathcal{C}$ be the unit clause at the top of the lattice such that $\mathcal{D} \in \rho_L^n(\mathcal{C})$. Then there exists m such that $\mathcal{D} \in \rho_M^m(\mathcal{C})$, where $m \leq n$ and $m, n \in \mathbb{N}^0$.*

Proof by induction. Let j_n be the last literal in $\mathcal{D}$. If $\mathcal{D}$ is a solution then per Theorem 5.1 j_n cannot be a dependent provider (i.e., $\mathcal{D}$ is a legal subsequence). Thus there are two cases to consider:

1. j_n is not a dependent consumer and there exists a macro $\mathbf{m}_b$ s.t. $\mathbf{m}_b = j_n$.
2. j_n is a dependent consumer and there exists a macro $\mathbf{m}_b$ s.t. $\mathbf{m}_b = j_i \dots j_n$.

Basis: Let $n = 1$. Since $n = 1$ there is only one body literal in $\mathcal{D}$ and thus j_n cannot be a dependent consumer (i.e., $\mathbf{m}_b = j_n$). Since j_n is the only body literal in $\mathcal{D}$ and $\mathcal{D} \in \rho_L^1(\mathcal{C})$ per Definition 4.6 all the input argument values ($+var$) of j_n are provided by the head of $\mathcal{D}$ and thus $\mathbf{m}_b$ is a legal subsequence given $\mathcal{C}$. Therefore $\mathcal{D} \in \rho_M^1(\mathcal{C})$.

Induction Step: Consider any $n > 1$ and let $\mathcal{E}_1 \in \rho_L^{n-1}(\mathcal{C})$ and $\mathcal{E}_2 \in \rho_M^{m-1}(\mathcal{C})$ such that $\mathcal{D} \in \rho_L(\mathcal{E}_1)$ and $\mathcal{E}_2 \subseteq \mathcal{E}_1$.

Assume the case when $\mathbf{m}_b = j_n$; in this case $\mathcal{E}_2 = \mathcal{E}_1$. For $\mathbf{m}_b$ to be a legal subsequence given $\mathcal{E}_2$ it is only required that at least one provider for every input argument of j_n is already in $\mathcal{E}_2$. All the providers of j_n can be added before $\mathbf{m}_b$ because $max\_prov(j_n) > max\_prov(k)$ for every provider k of j_n. Then a ρ_M-chain can be found so that $\mathbf{m}_b$ is a legal subsequence given $\mathcal{E}_2$. Thus $\mathcal{D} \in \rho_M^m(\mathcal{C})$, as claimed.

Assume the second case when $\mathbf{m}_b = j_i \dots j_n$; in this case $\mathcal{E}_2 \subset \mathcal{E}_1$ and all the literals in $\mathcal{E}_1 \setminus \mathcal{E}_2$ occur in $\mathbf{m}_b$. For $\mathbf{m}_b$ to be a legal subsequence given $\mathcal{E}_2$, at least one provider for every input argument value of j_l, $i \leq l \leq n$ in $\mathbf{m}_b$ is required to be in $\mathcal{E}_2$ or in $\mathbf{m}_b$. Since $max\_prov(\mathbf{m}_b) > max\_prov(k)$ for every provider k ($k < j_i$) of j_l, a ρ_M-chain can be found so that $\mathbf{m}_b$ is a legal subsequence given $\mathcal{E}_2$. Hence $\mathcal{D} \in \rho_M^m(\mathcal{C})$ as claimed.

This completes the proof of the induction step and of the theorem. $\square$

```
Input: Bottom clause $\bot = head \leftarrow 1, \ldots, n.$, mode declarations $M$.
Output: MSet (set of macros)
    1. For every literal $i$ $(1 \leq i \leq n)$ in $\bot$
        a) If $i$ is not a dependent consumer and $i$ is not a dependent provider
           Then add $i$ to MSet.
        b) If $i$ is not a dependent consumer and $i$ is a dependent provider
           Then
            i. dpM = Get_macros_of_dependent_provider($\bot, M, i$)
           ii. Add dpM to MSet.
    2. Sort MSet according to Definition 5.8.
    3. Output MSet
```

Algorithm 5.1: Obtaining the ordered set of macros MSet

5.3 Macros' Algorithms

We obtain the ordered set **MSet** of all macros from a given most specific clause and mode declarations using Algorithm 5.1. To obtain the macros based on a dependent provider *dp* in step 1b of Algorithm 5.1, **Get_macros_of_dependent_provider** (Algorithm 5.2) is called. In Algorithm 5.2, $\times$ means a special case of Cartesian product where the resulting set's elements are numerically ordered; $\mathbf{A}[j]$ represents the element j of set $\mathbf{A}$; and, $disaggregate(\mathbf{A}[j])$ is a function that separates the compound element $\mathbf{A}[j]$ into its component parts.

Let us explain Algorithm 5.2 in more detail. First, in steps 1 and 2, the set $\mathbf{Z}$ of all the consumers of *dp* is obtained. If a consumer k in $\mathbf{Z}$ is itself a dependent provider then a recursive call is performed to combine k with at least one of its consumers (step 3). In this way, Algorithm 5.2 generates macros which end with a non-dependent provider literal unless the dependent provider is as well a head provider (step 3(a)ii). Then *dp* is combined with each element in $\mathbf{Z}$ (step 5). After all the subsequences of literals starting with *dp* and ending with a non-dependent provider or with a head provider have been established, each subsequence of literals found is extended in step 6 with the providers required to make this subsequence satisfy the definition of a macro-operator (Definition 5.5). To constrain the number of macros generated, we discard any subsequence of literals as soon as it exceeds a user defined maximum number of literals in a clause.

Algorithm 5.2 differs from the algorithm presented in [79] in the recursive call in step 3. By doing this recursive call, the algorithm constructs macros with more than one dependent provider.

```
Get_macros_of_dependent_provider
Input: Bottom clause $\perp$, mode declarations $M$, dependent provider $dp$.
Output: dpM (set of macros based on $dp$)
    1. Let $\mathbf{Z}_1, \dots, \mathbf{Z}_n$ be the sets of consumers of $dp$ for $*var_1, \dots, *var_n \in dp$ (i.e., $\mathbf{Z}_j = \{k \in \perp \mid k > dp$ and $k$ consumes $*var_j \in dp\}$)
    2. Let $\mathbf{Z} = \bigcup_{j=1}^{n} \mathbf{Z}_j$
    3. For every literal $k$ in $\mathbf{Z}$
        a) If $k$ is a dependent provider Then
            i. $\mathbf{B}$ = Get_macros_of_dependent_provider($\perp$,$M$,$k$)
            ii. If $k$ is not a head provider Then Remove $k$ from $\mathbf{Z}$
            iii. Add $\mathbf{B}$ to $\mathbf{Z}$
    4. If $Z = \emptyset$ Then Output $\emptyset$
    5. Let $\mathbf{A} = \{dp\} \times \mathbf{Z}$
    6. While $\mathbf{A} \neq \emptyset$
        a) Let $a$ be $\mathbf{A}[0]$ And $\mathbf{Y} = \emptyset$
        b) Remove $a$ from $\mathbf{A}$
        c) Let $i$ be the first literal in $disaggregate(a)$
        d) For each literal $j \geq i + 1$ in $disaggregate(a)$
            i. For each $+var$ in $j$
                A. If $+var$ is not provided by the head And $+var$ needs a provider which is not in $a$ Then
                    Let $\mathbf{Y}_{j,+var}$ be the set of providers of $j$ for $+var$ (i.e., $\mathbf{Y}_{j,+var} = \{g \in \perp \mid g$ provides $+var$ to $j$ and ($g > i$ or $g$ is not a dependent provider )$\}$ )
                    Let $\mathbf{Y} = \mathbf{Y} \times \mathbf{Y}_{j,+var}$
        e) If $\mathbf{Y} \neq \emptyset$ Then
             Add $\{a \times \mathbf{Y}\}$ to $\mathbf{A}$
           Else Add $a$ to dpM
    7. Output dpM
```

Algorithm 5.2: Obtaining the macros based on a dependent provider

Example 5.1. Consider again the kinship example with the mode declarations on page 47 and the bottom clause on page 38. Algorithm 5.1 executes step 1b for literals $1, \dots, 4$ (they are all dependent providers). Since literals $5, \dots, 10$ are dependent consumers nothing is done for them.

For literal 1 in step 2 of Algorithm 5.2, we get $\mathbf{Z} = \{5, 6, 9, 10\}$; however, after step 3, $\mathbf{Z}$ is reduced to $\mathbf{Z} = \{9, 10\}$, because literals 5 and 6 are themselves dependent providers but no consumer of them was found in $\perp$. Then, in step 5 we obtain $\mathbf{A} = \{[1, 9], [1, 10]\}$. Since no provider of the literals 9 and 10 is needed, step 6 is executed only once per each literal and $\mathbf{dpM} = \{[1, 9], [1, 10]\}$ is returned.

For literals 2, 3, and 4, **Get_macros_of_dependent_provider** outputs $\emptyset$ in step 4. Thus, Algorithm 5.1 returns $\mathbf{MSet} = \{[1, 9], [1, 10]\}$.

Example 5.2. Now we illustrate how Algorithm 5.2 works with a more complex example. Suppose Get_macros_of_dependent_provider is given $\perp = h(+x) \leftarrow p(+x, *y, *z)$, $t(+x, -u)$, $o(+y, *w)$, $q(+x, -w)$, $r(+w, +z)$, $s(+u, +y)$, $m(+z)$, and $dp = p(+x, *y, *z)$.

```
(1) $\mathbf{Z}_y = \{o, s\}$, $\mathbf{Z}_z = \{r, m\}$
(2) $\mathbf{Z} = \{o, s, r, m\}$
(3) $o$ is a dependent provider **Then** $\mathbf{Z} = \{[o, r], s, r, m\}$
(5) $\mathbf{A} = \{p\} \times \{[o, r], s, r, m\} = \{[p, o, r], [p, s], [p, r], [p, m]\}$
(6) **While** $\mathbf{A} \neq \emptyset$
    (6a) $a = [p, o, r]$, $\mathbf{A} = \{[p, s], [p, r], [p, m]\}$
    (6e) $\mathbf{Y} = \emptyset$ **Then** $\mathbf{dpM} = \{[p, o, r]\}$
    (6a) $a = [p, s]$, $\mathbf{A} = \{[p, r], [p, m]\}$
    (6(d)iA) $\mathbf{Y}_{s,+u} = \{t\}$, $\mathbf{Y} = \{t\}$
    (6e) $\mathbf{Y} \neq \emptyset$ **Then** $\mathbf{A} = \{[p, r], [p, m], [p, t, s]\}$
    (6a) $a = [p, r]$, $\mathbf{A} = \{[p, m], [p, t, s]\}$
    (6(d)iA) $\mathbf{Y}_{r,+w} = \{o, q\}$, $\mathbf{Y} = \{o, q\}$
    (6e) $\mathbf{Y} \neq \emptyset$ **Then** $\mathbf{A} = \{[p, m], [p, t, s], [p, o, r], [p, q, r]\}$
    (6a) $a = [p, m]$, $\mathbf{A} = \{[p, t, s], [p, o, r], [p, q, r]\}$
    (6e) $\mathbf{Y} = \emptyset$ **Then** $\mathbf{dpM} = \{[p, o, r], [p, m]\}$
    (6a) $a = [p, t, s]$, $\mathbf{A} = \{[p, o, r], [p, q, r]\}$
    (6e) $\mathbf{Y} = \emptyset$ **Then** $\mathbf{dpM} = \{[p, o, r], [p, m], [p, t, s]\}$
    (6a) $a = [p, o, r]$, $\mathbf{A} = \{[p, q, r]\}$
    (6e) $\mathbf{Y} = \emptyset$ **Then** $\mathbf{dpM} = \{[p, o, r], [p, m], [p, t, s]\}$
    (6a) $a = [p, q, r]$, $\mathbf{A} = \emptyset$
    (6e) $\mathbf{Y} = \emptyset$ **Then** $\mathbf{dpM} = \{[p, o, r], [p, m], [p, t, s], [p, q, r]\}$
(7) $\mathbf{dpM} = \{[p, o, r], [p, m], [p, t, s], [p, q, r]\}$
```

Notice that the literals in a macro are always ordered according to their position in $\perp$. In this example the ordered set of all macros is $\mathbf{MSet} = \{\ t,\ q,\ [p, m], [p, t, s], [p, o, r], [p, q, r]\}$.

5.3.1 Correctness of the Algorithms

We now show that the algorithms work as they should; that is, every macro constructed by them is a correct macro (i.e., the macro satisfies Definition 5.5).

Theorem 5.3. *Let dp be a dependent provider which is not a dependent consumer. Then Algorithm 5.2 with dp as input returns the set* **dpM** *of macros based on dp such that every macro in* **dpM** *satisfies Definition 5.5.*

Proof by contradiction. Assume $\mathbf{m}_b = j_{i+1} \dots j_m$ is a macro in **dpM** generated by **Get_macros_of_dependent_provider** and $\mathbf{m}_b$ does not satisfy Definition 5.5. That is, either (1) there is no subsequence $j_1 \dots j_i$ such that $\mathbf{m}_b$ is legal given $j_1 \dots j_i$, or (2) a proper subsequence of $\mathbf{m}_b$ is legal given $j_1 \dots j_i$.

1. Assume the former is true, then (see Definition 5.4) there should be a dependent provider in $\mathbf{m}_b$ without a consumer, or there should be a consumer j_k $(i + 1 < k \leq m)$ in $\mathbf{m}_b$ missing a provider p which cannot be in a subsequence $j_1 \dots j_i$. However, after steps 1–5 every dependent provider in $\mathbf{m}_b$ has at least one consumer or is a head provider; and, after step 6 every required provider p is either in $\mathbf{m}_b$ or $p < j_{i+1}$ so that p can be in a subsequence $j_1 \dots j_i$. Thus, $\mathbf{m}_b$ is legal given $\emptyset$ or there exists a subsequence $j_1 \dots j_i$ such that $\mathbf{m}_b$ is legal given $j_1 \dots j_i$, which contradicts our first assumption.
2. Assume now the second case is true, then we could remove a literal l from $\mathbf{m}_b$ such that it would still be legal given $j_1 \dots j_i$. However, since step 5 adds only one consumer per dependent provider and step 6 adds only one provider for every $+var$, if l is in $\mathbf{m}_b$, l is either a consumer of a dependent provider and thus $\mathbf{m}_b$ without l is not legal given $j_1 \dots j_i$, or l is a provider $> j_{i+1}$ needed by a consumer in $\mathbf{m}_b$ and thus $\mathbf{m}_b$ without l is not legal given $j_1 \dots j_i$ since l cannot be in a subsequence $j_1 \dots j_i$. Thus no proper subsequence of $\mathbf{m}_b$ is legal given $j_1 \dots j_i$, which contradicts our second assumption and completes the proof.

□

Theorem 5.4. *Let* $\perp$ *be a bottom clause and M a set of mode declarations. Then Algorithm 5.1 with* $\perp$ *and M as input returns the ordered set* **MSet** *of macros such that every macro in* **MSet** *satisfies Definition 5.5.*

Proof. It is clear that every literal l added in step 1a fulfills Definition 5.5 since l is not a dependent provider and every provider of l is before l in $\perp$ so that there is a subsequence $j_1 \dots j_i$ such that l is legal given this subsequence. In addition, by Theorem 5.3 all the macros added in step 1b satisfy Definition 5.5. □

5.3.2 Completeness of the Algorithms

In this section we show that Algorithm 5.1 generates every possible macro given a bottom clause and a set of mode declarations.

Theorem 5.5. *Let $\perp$ be a bottom clause and M a set of mode declarations. Then Algorithm 5.1 with $\perp$ and M as input returns the ordered set* **MSet** *of macros such that no correct macro which can be obtained from the given input is missing in* **MSet***.*

Proof. The algorithm to obtain **MSet** generates the macros traversing the bottom clause so that it must eventually consider every literal in $\perp$. Thus every one-literal macro is added to **MSet** in step 1a and **Get_macros_of_dependent_provider** (see Algorithm 5.2) is executed for every dependent provider which is not a dependent consumer.

Get_macros_of_dependent_provider generates macros composed of two or more literals. In this case, for a macro based on a dependent provider *dp* to be missing in **MSet**, it is necessary that either a consumer of *dp* is not considered or some combination of providers is not taken into account. However, steps 1 and 2 ensure that every consumer of *dp* is considered for inclusion in a macro, and step 6 generates a macro for every possible combination of available providers. Thus, every macro which can be obtained from the given input is in **MSet**. □

5.3.3 Identifying the Dependent Providers

To be able to use ρ_M and the Algorithms 5.1 and 5.2, one needs to identify the dependent providers on the application domain. This is easy for domains with predicates denoting mathematical functions such as **diff/3** which computes the absolute difference between two numbers, because those predicates ought to be dependent providers. However, in some cases, the meaning of the predicates is not as clear such that the dependent providers can be easily identified. Fortunately, an automatic test can be done to identify the dependent providers. Basically, for every body predicate with output arguments, one obtains how many different instantiations of their input variables are in $B \cup E$ such that there is at least one instantiation of their output variables. This is done using the Prolog predicate **setof/3**, which returns the non-empty set of all possible ground bindings of one or more given variables such that a given query is satisfied. If there are no ground bindings of the given variables such that the query is satisfied then **setof** fails.

Let us illustrate this test using the mutagenesis domain where the target predicate is **active/1** whose mode declaration is **modeh: active(+drug)**. Assume we want to find out whether the predicate **lumo** whose mode declaration is **modeb: lumo(+drug,-energy)** is a dependent provider with respect to background knowledge B and training examples E. Then we execute the following Prolog query:
setof((Drug), (Energy)^(drug(Drug), lumo(Drug,Energy)), Set),

length(Set,NumEx).
This query asks how many different ground bindings for the input variable **Drug** are there such that there exists at least one ground binding for the output variable **Energy**. The instantiations of the input variable **Drug** are given by the elements of type **drug** in B. If this query instantiates **NumEx** to a value equal to the total number of elements of type **drug** in B, the background predicate, in this case **lumo**, is a dependent provider.

5.3.4 Mio

We implemented the literal-based approach and the macro-based approach (including Algorithms 5.1 and 5.2) in the system Mio[3]. Mio is an example-driven covering system which performs a top-down search in the hypothesis space $H^{\perp}$ (see Definition 4.5). Three search strategies are included in Mio: IDA*, beam-search and hill-climbing. In this chapter, we concentrate in IDA* because macros are first shown to work within exhaustive search; hill-climbing and beam-search are treated in the next chapter.

IDA* in Mio is guided by the number of literals needed to obtain an I/O-complete clause. When using IDA*, Mio evaluates a clause $\mathcal{C}$ using the evaluation function $eval(\mathcal{C})$ shown in Equation 5.1. In Equation 5.1 $\mid \mathcal{C} \mid$ is the size of $\mathcal{C}$, pos is the number of positive examples covered by $\mathcal{C}$ and neg is the number of negative examples covered by $\mathcal{C}$.

$$eval(\mathcal{C}) = \frac{pos}{\mid \mathcal{C} \mid} - neg \tag{5.1}$$

The idea behind Equation 5.1 is to penalize inconsistent clauses and, given two clauses $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same coverage, to favour the smallest one.

Besides the use of macros, Mio distinguishes itself from other systems in that it performs parallel search, selects stochastically the examples to guide the search, and enforces type strictness (Definition 4.3). The stochastic example selection and parallel search are done to avoid that the order in which the examples are given to the system affects the learning results[4]. As already mentioned in Section 4.1, type strictness is done to discard meaningless hypotheses.

[3]Mio's user manual is provided in Appendix A.
[4]Parallel search is discussed in Chapter 7.

5.4 Empirical Evaluation

Until this point, we have shown that a learning system using ρ_M explores, at least in theory, less clauses than ρ_L when learning in application domains with dependent providers. In this section we empirically analyze the performance of the macro-based method in comparison with the literal-based method. The goal of our empirical evaluation was to find out:

a) What is the search space reduction obtained by using the macro-based ρ_M, and

b) how this reduction reflects on the running time of the learning system.

In addition, we compare Mio (using IDA*) with Progol.[5] We decided to use Progol in these experiments because both learning systems perform exhaustive search and construct the bottom clause using inverse entailment. In addition, as mentioned in Section 3.3, Progol is the most commonly used system in ILP publications of the last 7 years. Mio differs from Progol in several aspects such as including parallel search, hill-climbing search and beam-search; enforcing type strictness; selecting stochastically the examples to guide the search; and supporting the use of macros and active inductive learning[6].

5.4.1 Application Domains

We carried out experiments on the following datasets.

1. Chess moves[7]. This dataset contains 180 positive and 17 negative examples of valid chess moves for five pieces: king, queen, rook, bishop and knight. The learning task is to learn what are correct moves for these pieces.
2. Eastbound trains[8]. One has to determine the direction (east or west) of trains based on their attributes. This dataset contains 42 positive and 19 negative examples.
3. Student loan[9]. This dataset was used in [77] and consists of 643 positive and 357 negative examples. The learning problem is to discriminate between individuals who are not required to pay back an educational loan and those who must pay.

[5]Comparisons with other systems are presented in Chapter 6.
[6]Active inductive learning is discussed in Section 8.2.
[7]Part of this dataset is contained in the distribution package of CProgol 4.4.
[8]To generate the examples we used the Random Train Generator available at www.doc.ic.ac.uk/~shm/Software/GenerateTrains/. The dataset we used is available upon request.
[9]This dataset is available at the UCI repository [3].

Dataset	Dependent Providers
Chess moves	**diff/3**
Eastbound trains	**has_car/2**, **infront/3**
Student loan	**longest_absence_from_school/2**, **enrolled/3** (96%)
Mutagenesis	**lumo/2**, **logp/2**, **bond/4**, **nitro/2**, **benzene/2** (99%), **ring_size_6/2** (99%)
Mesh design	**neighbour/2** (98%)
Traffic	**secciones_posteriores/2** (74%)

Table 5.1: Dependent providers declared for application domain

4. Mutagenesis[10]. This is an ILP benchmark dataset described in [100] with 188 compounds (125 positive and 63 negative examples). The mutagenesis problem deals with the prediction of the mutagenic activity of small, heterogeneous molecules.
5. Mesh design[11]. The finite element method is used to analyze stresses in physical structures by approximating them with a mesh model. The learning problem is to determine the appropriate number of elements N on an edge. This dataset contains information about the edges of ten different structures and is described in [25].
6. Traffic problem detection. This dataset was used in [32] and consists of 256 examples of traffic situations in road sections in the city of Barcelona. The task is to classify such situations as accident, congestion or non-critical section.

Table 5.1 shows which background predicates were declared as dependent providers for each domain. In this table, a percentage x following a predicate indicates that the property or relation denoted by the predicate does not hold for every input argument values but holds for $x\%$ of the input argument values (as explained in Section 5.3.3).

5.4.2 Experiment Design

In the experiments, we set the maximum number of clauses explored per search by Mio to 70000 so that the searches were not interrupted by reaching a clause limit. For Progol, we follow the recommendations in [73] and set the maximum

[10]Available at web.comlab.ox.ac.uk/oucl/research/areas/machlearn/mutagenesis.html.
[11]This dataset is available at www.mlnet.org.

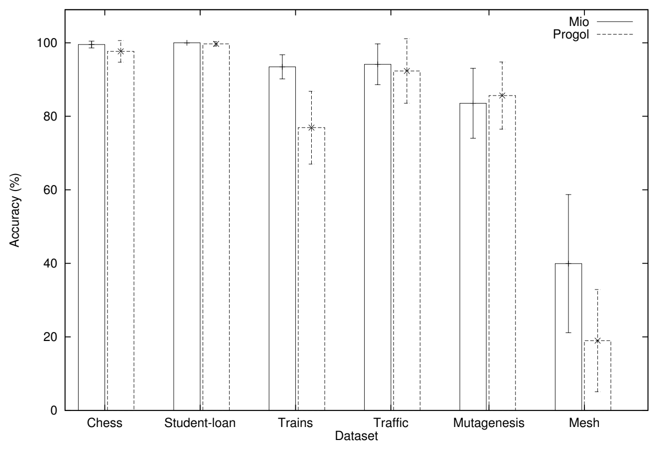


Figure 5.2: MioM (left) and Progol's accuracies per dataset

number of nodes to 1.6 times the number of examples in E when this number is greater than the default value (200). This was done for every dataset, except for mutagenesis and eastbound trains, where the settings indicated by the author of Progol were used. Appendix B contains the settings and mode declarations used for each system per dataset.

To learn and evaluate the results, we used 5-fold-cross-validation for the first three datasets (because of Mio's small standard deviation), 10-fold-cross-validation for mutagenesis and traffic, and cross-validation-leave-one-out for mesh design (i.e., the systems learned from nine structures and the theories were evaluated in the tenth structure). In the mesh dataset, negative examples were introduced for learning but they are not included in the test data, because the accuracy obtained when including them is misleadingly high. Mio was run twice on every dataset: once using ρ_L (MioL) and once using ρ_M (MioM).

5.4.3 Results

Figure 5.2 shows the average accuracy and standard deviation of MioM and Progol per application domain. In the chess, trains, student loan and mesh design domains, the accuracy of Mio is statistically higher than that of Progol (t-significance level $\alpha = 0.01$); while there is no statistically significant difference between Mio and Progol's accuracies in the mutagenesis and traffic domains.

As Theorem 5.2 asserts, in domains with dependent providers, MioL and MioM obtain the same theory, and thus, their accuracy is the same. However, in mesh design and the traffic problem, sometimes there are clauses which differ between

	Average nodes/search			Average run time		
Dataset	**MioL**	**MioM**	**Progol**	**MioL**	**MioM**	**Progol**
Chess	79	29	33	8.2s	5.2s	1.2s
Trains	4862	2754	1230	3.3m	2.5m	54.2s
Student-Loan	72	33	60	14s	8.3s	17s
Mutagenesis	42919	11191	20000	6h30m	1h54m	1h18m
Mesh	23854	3334	7900	18h30m	4h53m	13h55m
Traffic	479	175	420	6.4m	2.4m	3.2m

Table 5.2: Search space explored and run time per system per dataset

Dataset	**Search space reduction**	**Speed-up ratio**
Chess	63%	1.6
Trains	43%	1.3
Student-Loan	54%	1.7
Mutagenesis	74%	3.4
Mesh	86%	3.8
Traffic	63%	2.7
Average	$64 \pm 14\%$	2.4 ± 0.95

Table 5.3: Comparison of MioM against MioL

the theories learned by MioM and MioL. In some of these cases, the accuracy of MioM is slightly higher; e.g., in the traffic dataset MioM has an average accuracy of $94.2 \pm 5.5\%$ versus $93.9 \pm 5.5\%$ obtained by MioL, and in the mesh dataset MioM obtains an accuracy of $39.9 \pm 18.8\%$ while MioL obtains $38.1 \pm 18.3\%$. This indicates that the macro-based ρ_M can also be used in domains with providers which succeed for most of the input argument values as long as a consumer of these providers is also available.

Table 5.2 shows the average number of nodes (clauses) expanded per search and the run time per system per dataset (all the experiments were run on a 500 MHz Sun Blade 100 with 128 MB of RAM). In Table 5.3 the column *speedup ratio* shows the geometrical ratio of MioL's average run time t_1 to MioM's average run time t_2 (i.e., t_1/t_2). The search space reduction obtained by using ρ_M compared with ρ_L is calculated with the formula $(1 - s_2/s_1) * 100$, where s_1 and s_2 are the average number of nodes explored per search by MioL and MioM, respectively.

ρ_M evaluates on average $64 \pm 14\%$ less clauses than ρ_L. In fact, in every domain MioM considers statistically less clauses than MioL (t-significance level $\alpha = 0.005$). With this search space reduction, MioM is on average 2.4 times faster

Dataset	Ave. No. literals in $\perp$	Ave. No. macros in MSet	Ave. macros construction time (msec)
Chess	20 ± 5	29 ± 10	2.5 ± 1.1
Trains	33 ± 8	202 ± 68	19.8 ± 7.4
Student-Loan	8 ± 2	6 ± 1	0.7 ± 0.6
Mutagenesis	210 ± 43	432 ± 104	29.7 ± 10.8
Mesh	97 ± 35	92 ± 49	9.7 ± 5.4
Traffic	34 ± 10	21 ± 6	2.2 ± 0.9

Table 5.4: Average macros construction time per dataset

than MioL. Note that this improvement is obtained without further restricting the hypothesis space; i.e., MioM and MioL search for solutions in the same hypothesis space but MioM considers only legal sequences of literals. MioM's run time is comparable to that of Progol even though Mio was allowed to consider a larger number of clauses per search than Progol.

To determine the overhead caused by the macros construction algorithms in Mio's run time, we measured the average construction time per dataset. Table 5.4 shows per dataset the average size of the bottom clause, the average number of macro-operators constructed, and the average macros construction time in milliseconds. On average, there is a time overhead of 0.1 milliseconds per macro automatically constructed. Thus, the macros construction time is a minor overhead in the performance of the system.

5.5 Related Work

Reviewers of our work have pointed out that the definition of macros reminds them of k-local clauses, a language bias[12] proposed by Cohen [14]. Cohen introduced k-local clauses as an alternative restriction to determinate clauses which still leads to a pac-learnable language[13]. In this bias, only clauses of locality k or less are considered, where the locality of a clause is the size of the largest set of literals which contain either a free (local) variable X or some free variable

[12]A bias refers to any criteria used by a learning system for preferring one clause over another when constructing a theory, other than strict consistency with training data.

[13]Pac stands for probably approximately correct, which means that a system can probably learn from a polynomial number of training examples a theory that is approximately correct (see [65], Chapter 7).

influenced by X (this set is called the *locale* of X)[14]. However, contrary to the clauses generated by a macro-based refinement operator, a k-local clause might not be a legal subsequence of literals, in this case, it does not need to be considered because it cannot be a solution. For example, suppose we search for 3-local clauses. In this case, a clause whose body consists of one dependent provider is considered. However, as discussed before, we know that such a clause cannot be a solution. A macro-based refinement operator, on the other side, does not generate this kind of clauses.

Another work reminiscent of macros is first-order feature construction for individual-centered domains by Flach and Lavrač. This approach is used in 1BC [37] to restrict the search space, and in LINUS [57, 58] and RSD [60] to propositionalize the input data. *Individual-centered* domains [35] are domains in which there is a clear notion of individual; i.e., an example refers to a single individual and the target concept is a property of a set of individuals. The relation between individual-centered first-order feature construction and macros becomes more apparent if the macro generation is seen as first-order feature construction and macros are considered first-order features used by the refinement operator to construct the body of a clause. In both approaches, feature construction and macros, the key idea is to use provider-consumer iterations of existential variables among the literals as basis to construct the features or, respectively, the macros.

To establish the differences between first-order features and macros, let us relate the terms used in [37, 58] to macros' terminology. *Structural predicates* are restricted to represent binary relations between complex structures (or types) with one of their parts. Since dependent providers are n-ary predicates which can denote any relation or function, structural predicates can be seen as a proper subset of binary dependent providers. Structural predicates are similar to structural literals defined by Zucker and Ganascia [108] but structural predicates relax the condition of transitivity (see Definition 2.28). Our dependent providers relax in addition the requirement of antisymmetry. *Properties* or *utility predicates* are equivalent to consumers, and *features* are legal subsequences of literals (i.e., a feature satisfies Definition 5.3) where head providers are not allowed. Macros extend individual-centered feature construction by allowing head providers and n-ary dependent providers. In addition, macros can be applied to non-individual-centered domains such as the kinship example used in this chapter or for program synthesis tasks.

[14]Let X, Y and Z be free variables. According to [14], X influences Y if they appear in the same literal, or if X occurs with Z in the same literal and Z appears in some other literal together with Y. Note that the relation *influences* does not imply a provider-consumer relation.

Macros have four advantages with respect to other ILP approaches to restrict the size of the search space. First, contrary to *ij*-determinate literals (see Definition 3.8), macros are suitable for a large variety of application domains and do not require properties such as determinacy (Definition 3.7) in the application domain which are not present in many interesting problems. Second, compared with *k*-local clauses, macros prevent the system from considering unneeded clauses; i.e., clauses which cannot belong to an appropriate theory. Third, macros are automatically constructed and do not imply extra work for the user. Finally, contrary to most language biases, macros guarantee that only unneeded clauses are discarded.

Macros can also be considered a formalism to specify language bias (such as MOBAL's schemata [52] or ADG [13]). Within ADG, Cohen [12] uses the term *lazy macros* to denote a method for completing a given antecedent description grammar, but this is unrelated to our macros' definition. Macros are also related to approaches used to solve the shortsightedness of a learning system using hill-climbing search but these approaches are surveyed in Chapter 6.

5.6 Summary

In this chapter we precisely develop the use of automatically constructed macros as a formal technique to reduce the search space defined by a downward refinement operator. Macros are suitable for domains with dependent providers or providers which succeed for most of the input argument values, and they do not add incompleteness to the search. By using a macro-based refinement operator, we discard a significant number of clauses which cannot belong to an adequate theory. Thus, a reduction in the search space is obtained which results in shorter run-times. This approach was implemented in Mio, an example-driven covering system. Experiments on six application domains show that an average search space reduction of 64% obtained using ρ_M produces on average a 2.4 fold speedup of the learning process.

In addition, we provided algorithms for constructing the macros based on a set of mode declarations and a bottom clause. These algorithms are shown to be complete and correct.

6 Hill-Climbing Search Using Macros

In the previous chapter we introduced macros and explained how they reduce the search space explored when exhaustive search (e.g., IDA*) is used. However, even with a search space reduction, considering all possible alternatives at each level of the refinement graph (i.e., performing exhaustive search) can be too inefficient because the branching factor of most multirelational learning problems is very large. Consequently, search strategies which only take a limited number of alternatives at each level are typically applied. Among these search strategies, hill-climbing search, which takes only one (the best) alternative at each level, is the most commonly used search algorithm in ILP.[1] However, a well known problem of systems using hill-climbing search is their *myopia*; i.e., the search algorithm might be unable to correctly assess the quality of a refinement and end up with a non-optimal theory.

This chapter examines the use of macro-operators as an alternative approach to mitigate hill-climbing's myopia. As shown in Chapter 5, macro-operators combine dependent providers (see Definition 5.1) with their consumers, and a macro-based refinement operator ρ_M (Definition 5.9) only generates clauses which are legal subsequences of literals (Definition 5.3). Hill-climbing's shortsightedness usually occurs because hill-climbing search does not consider the existence of dependent providers and the inability of the evaluation function (Definition 3.12) to deal properly with this kind of literals. For example, consider the eastward trains domain where the learning task is to find a theory to classify the trains according to their traveling direction (east or west). One of the relations in this domain is the structural relation between one train and its cars denoted by the predicate **hasCar**/2. Since every train is composed by cars, **hasCar** is a dependent provider and does not differentiate between examples of trains belonging to different classes. Thus, the evaluation value of clauses that contain

[1]The reader is referred to [40] for a list of ILP systems and their search strategies.

hasCar as their last literal is low and they are not selected by hill-climbing (i.e., **hasCar** does not occur in any learned clause). In this case, hill-climbing search may end up with a non-optimal clause or hit a dead end. The fact that hill-climbing cannot "see" that **hasCar** combined with other relations may yield a solution (Definition 4.8) is called *myopia*.

The benefit derived from macro-operators in hill-climbing is that the evaluation function is only applied to legal subsequences of literals, which has the advantage that the quality of the clauses can be more accurately assessed since they have discriminative power and can in fact be a solution. For instance, in the eastward trains domain with the macro-based approach, **hasCar** is always added to a clause with one of its consumers so that the evaluation function is able to more accurately estimate its importance.

In the next section, our hill-climbing search algorithm is presented. In Section 6.2, we use this algorithm to illustrate hill-climbing's myopia problem. Section 6.3 explains how macros alleviate the shortsightedness of hill-climbing. Section 6.4 presents a detailed comparative study on approaches to reduce the myopia problem of hill-climbing search in multirelational learning. This study involves fixed-depth lookahead, template-based lookahead, beam-search, determinate literals and macros. Our results show that, with the exception of beam-search, a hill-climbing learner using macros reports significantly lower classification error than systems using other approaches. Finally, Section 6.5 discusses related work.

6.1 A Hill-Climbing Search Algorithm

With the purpose of having a general search algorithm able to perform various search strategies, we formulate Algorithm 6.1. This algorithm performs a top-down search and uses an example to guide the search for hypothesis. It receives two user-defined parameters, s and b, which indicate the amount of lookahead and the beam width, respectively. The default values ($s = 1$ and $b = 1$) correspond to hill-climbing search. An s with value of x means that step 2(b)ii in Algorithm 6.1 adds to $\mathbf{R}$ the set of all refinements $\mathbf{W}$ obtained by x or less applications of ρ to $\mathcal{D}$; that is, the system performs an x-step lookahead. With a value of b greater than one ($1 < b \ll \infty$), beam-search is obtained. In addition, ρ can be either a literal-based refinement operator (ρ_L) or a macro-based refinement operator (ρ_M). As in Chapter 5, when ρ_L is employed we refer to Mio as MioL and when ρ_M is used as MioM.

To evaluate the quality of a clause $\mathcal{C}$, Algorithm 6.1 uses the evaluation function shown in Equation 6.1 which is based on the information gain heuristic [41]. In Equation 6.1, *pos* is the number of positive examples covered by $\mathcal{C}$; *neg* the

```
Top_Down_Search
Input: The top of the lattice $\top$, $B$, $E$, $b$, $s$
Output: Either a clause $\mathcal{C}$ or $\emptyset$
    1. $\mathbf{S} = \{\top\}$ /* Ordered set of clauses to consider*/
    2. While a clause $\mathcal{D} \in \mathbf{S}$ can be refined or search resources are not
       exhausted
        a) $\mathbf{R} = \emptyset$ /* Set of refinements */
        b) For every clause $\mathcal{D} \in \mathbf{S}$ that can be refined
            i. $\mathbf{W} = \bigcup_{d=1}^{s} \rho^d(\mathcal{D})$
            ii. $\mathbf{R} = \mathbf{R} \cup \mathbf{W}$
        c) Sort $\mathbf{R}$ according to $eval$
        d) Let $\mathbf{R}_b$ be the best $b$ refinements in $\mathbf{R}$
        e) $\mathbf{S} = \mathbf{R}_b$
    3. Let $\mathcal{C}$ be the best evaluated clause in $\mathbf{S}$
    4. If $eval(\mathcal{C}) \geq$ minimum value accepted and $\mathcal{C}$ covers enough positive
       and few enough negative examples in $E$
        a) Then Return $\mathcal{C}$
        b) Else Return $\emptyset$
```

Algorithm 6.1: A search algorithm to perform literal-based or macro-based hill-climbing and beam-search with or without lookahead

number of negative examples covered by $\mathcal{C}$; $|E^+|$ the total number of positive examples; $|E^-|$ the total number of negative examples; $|\mathcal{C}|$ the number of body literals in $\mathcal{C}$ (or 1 if $\mathcal{C}$'s body is empty); $\top$ refers to the unit clause at the top of the refinement lattice, and $IC(\mathcal{D})$ returns the information content of any clause $\mathcal{D}$. $IC(\mathcal{C})$ is calculated by $-log_2(\frac{pos}{pos+neg})$.

$$eval(\mathcal{C}) = \frac{\frac{pos+(|E^-|-neg)}{|E^+|+|E^-|} * (IC(\top) - IC(\mathcal{C}))}{|\mathcal{C}|}. \tag{6.1}$$

From an information theory perspective, $IC(\top) - IC(\mathcal{C})$ can be interpreted as the reduction due to $\mathcal{C}$'s literals in the total number of bits needed to encode the classification of an arbitrary positive example. In Equation 6.1, we weight $\mathcal{C}$'s information gain with its accuracy, because two clauses may have equal information gain but different accuracy and we want the evaluation function to reward accuracy. In addition, given two different clauses with the same information gain and accuracy, Equation 6.1 assigns a higher value to the shortest one.

In the subsequent sections, every reference to Mio pertains to Mio using Algorithm 6.1, unless otherwise stated.

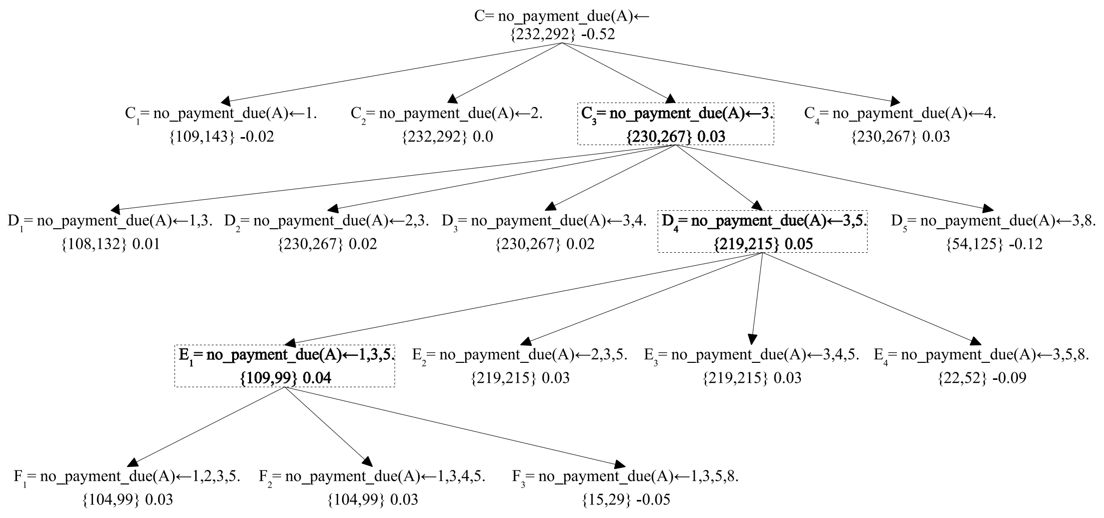


Figure 6.1: Search space explored by hill-climbing using ρ_L. Below each clause one can see between braces the number of positive and negative examples covered by the clause followed by the clause's heuristic value obtained by Equation 6.1. A square enclosing a clause indicates the clause chosen by hill-climbing at each iteration.

6.2 Myopia of Hill-Climbing: An Example

To illustrate the myopia problem of hill-climbing using Algorithm 6.1, consider the student loan domain where the system has to induce a theory to classify individuals into those who are not required to pay back an educational loan and those who must pay. Assume we set the maximum length of the clauses to four body literals[2] and that in one iteration of the covering algorithm the following bottom clause is derived from a given example e.

no_payment_due(A) ← -1- male(A), -2- longest_absence_from_school(A,I0), -3- enrolled(A,School1,I1), -4- enrolled(A,School2,I2), -5- gte(I1,3), -6- gte(I0,4), -7- gte(I2,9), -8- lte(I1,3), -9- lte(I0,4), -10- lte(I2,9).

Recall from Chapter 5, that the numbers which precede the literals in the bottom clause indicate the position of each literal in $\perp$ and they are used to refer to the literals.

Let us explain how Algorithm 6.1 with b and s equal to one and ρ being ρ_L works. In the first iteration, after step 2b, the set **R** contains four refinements

[2]The corresponding mode declarations can be seen in Appendix B.

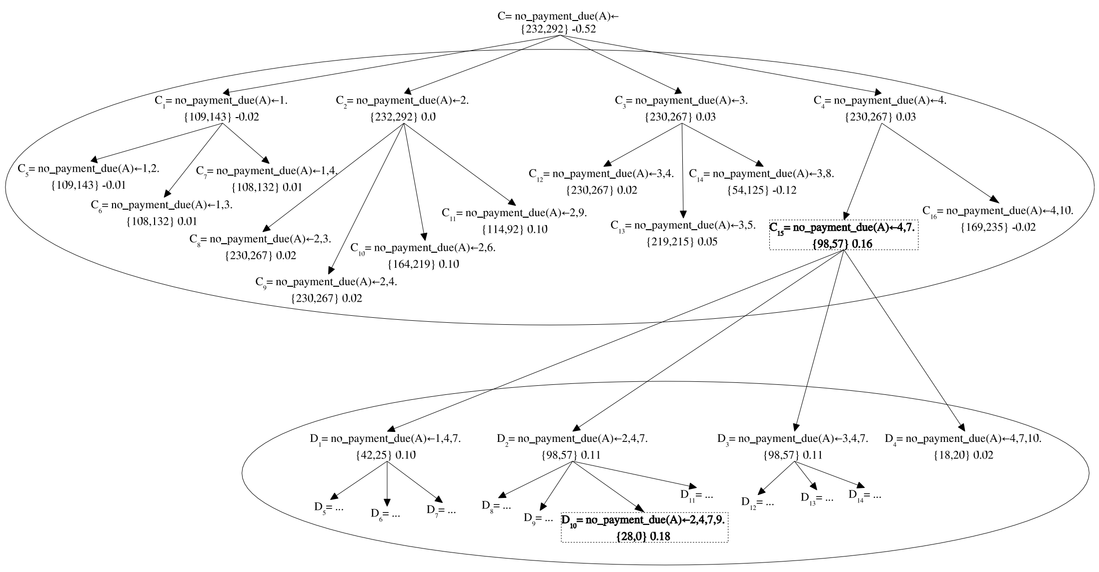


Figure 6.2: Search space explored by hill-climbing using ρ_L with 2-step lookahead. The ellipses inclose the clauses generated by ρ_L per iteration (Algorithm 6.1, step 2b).

of the clause $\mathcal{C}$ =**no_payment_due(A)**← (these refinements are $\mathcal{C}_1, \ldots, \mathcal{C}_4$ in Figure 6.1). From these refinements, $\mathcal{C}_3$ and $\mathcal{C}_4$ are the best evaluated clauses by Equation 6.1. Assume that, in step 2d, $\mathcal{C}_3$ is chosen. Then, after three more iterations, hill-climbing hits a dead end with three clauses in **S** which cannot be refined ($\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ in Figure 6.1), and since $\mathcal{F}_1$ does not satisfy the criterion in step 4, the algorithm returns $\emptyset$. Hill-climbing fails to find a solution because the literals 2, 3 and 4 are dependent providers (i.e., they do not have discriminative power) and the evaluation function is unable to correctly assess their quality when added by themselves to a clause. The refinements which lead to a solution in this example are $\mathcal{C}_2$ and $\mathcal{C}_4$ and the solution is the clause **no_payment_due(A)** ← 2,4,7,9.

In this case, 2-step lookahead solves the myopia of hill-climbing and finds a solution (see Figure 6.2); however, employing 2-step lookahead has the drawback that several non-admissible clauses (e.g. clauses which contain a dependent provider as the last literal such as $\mathcal{C}_5$ in Figure 6.2) are considered; and, we know that the evaluation function cannot adequately estimate the value of such clauses. Furthermore, as shown in Section 6.4, fixed-depth lookahead may incur in significantly longer run times without gain in accuracy. In the next section, we illustrate how a macro-based approach finds a solution for this learning problem.

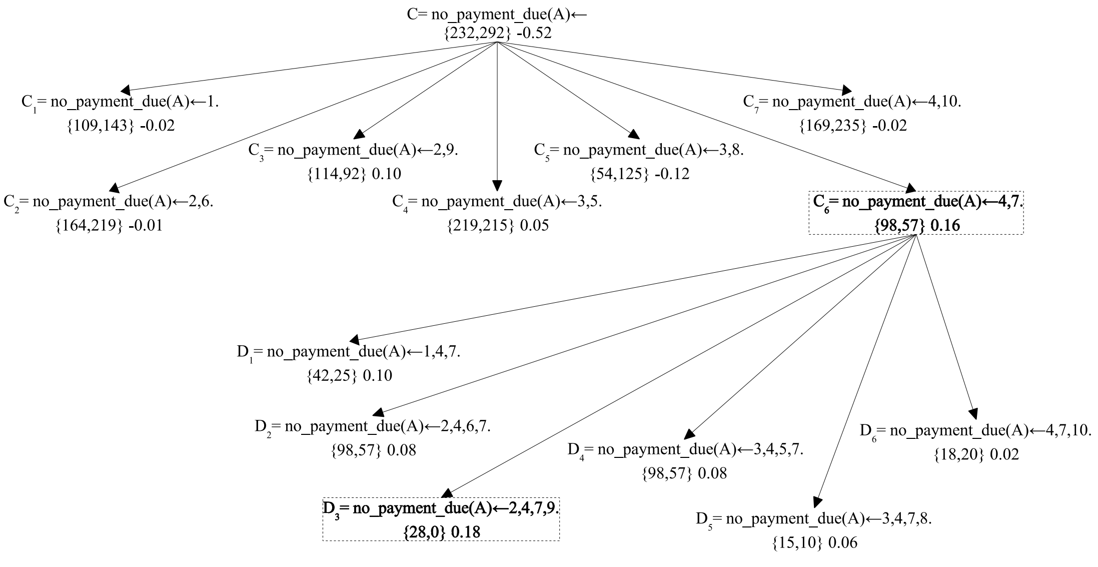

C= no_payment_due(A)←
{232,292} -0.52
C1= no_payment_due(A)←1.
{109,143} -0.02
C2= no_payment_due(A)←2,6.
{164,219} -0.01
C3= no_payment_due(A)←2,9.
{114,92} 0.10
C4= no_payment_due(A)←3,5.
{219,215} 0.05
C5= no_payment_due(A)←3,8.
{54,125} -0.12
C6= no_payment_due(A)←4,7.
{98,57} 0.16
C7= no_payment_due(A)←4,10.
{169,235} -0.02
D1= no_payment_due(A)←1,4,7.
{42,25} 0.10
D2= no_payment_due(A)←2,4,6,7.
{98,57} 0.08
D3= no_payment_due(A)←2,4,7,9.
{28,0} 0.18
D4= no_payment_due(A)←3,4,5,7.
{98,57} 0.08
D5= no_payment_due(A)←3,4,7,8.
{15,10} 0.06
D6= no_payment_due(A)←4,7,10.
{18,20} 0.02

Figure 6.3: Search space explored by hill-climbing using ρ_M. A square enclosing a clause indicates the clause chosen by hill-climbing at each iteration.

6.3 How Macros Alleviate the Myopia of Hill-Climbing

Let us illustrate how macros alleviate the myopia of hill-climbing using the example above. Since every body literal applying the predicates **longest_absence_from_school** and **enrolled** introduces existential variables to a clause and succeeds for any combination of input argument values, these literals are dependent providers. Seven macros are built by the macros generation algorithm described in Section 5.3 with the bottom clause listed above and the mode declarations given in Appendix B as input. These seven macros are $\mathbf{MSet} = \{1, [2,6], [2,9], [3,5], [3,8], [4,7], [4,10]\}$.

As depicted in Figure 6.3, in the first iteration, macro-based hill-climbing (i.e., Algorithm 6.1 with b and s equal to one and ρ being ρ_M) evaluates seven refinements obtained by adding to $\mathcal{C}$ =**no_payment_due(A)**← the generated macros. From those refinements, $\mathcal{C}_6$ obtains the highest heuristic value and is selected in step 2d. In the second iteration, in step 2b, $\mathcal{C}_6$ is refined by adding the available macros, and in step 2d, $\mathcal{D}_3$ is selected. Since $\mathcal{D}_3$ cannot be further refined, the algorithm terminates and returns $\mathcal{D}_3$, which is a solution and is then added to the final theory. $\mathcal{D}_3$ corresponds to the following solution.

$$\mathcal{D}_3 = \textbf{no\_payment\_due(A)} \leftarrow \textbf{longest\_absence\_from\_school(A,I0),}$$
$$\textbf{enrolled(A,School2,I2), gte(I2,9), lte(I0,4).}$$

Macros alleviate hill-climbing's myopia problem because Equation 6.1 is applied to clauses whose quality can be more accurately assessed because they have discriminative power and can be a solution. By using macros, an automatically adjusted variable-depth lookahead is performed where only legal subsequences of literals are considered.

6.4 Empirical Evaluation

In this section we empirically analyze how effective macros are to alleviate the shortsightedness of hill-climbing search compared with other approaches against myopia. Specifically, we compared macros with fixed-depth lookahead, beam-search, determinate literals and template-based lookahead. These approaches were compared in terms of classification error and run time.

We performed experiments on the six application domains described in the previous chapter (page 55) which are chess moves, student loan, eastward trains, mesh design, traffic problem detection, and mutagenesis. The same as in Chapter 5, we used 5-fold-cross-validation for the first three datasets, 10-fold-cross-validation for mutagenesis and traffic, and cross-validation-leave-one-out for mesh design.

6.4.1 Comparing Macros with Template-based Lookahead and Determinate Literals

For the comparison with template-based lookahead and determinate literals, we carried out experiments with FOIL (version 6.4) and TILDE (contained in ACE 1.1.15), respectively. We defined the templates needed by TILDE to perform lookahead based on the macros created by MioM. This was done for every dataset except for mutagenesis and mesh where the templates indicated by the author of TILDE were used [4]. MioM was run with $s = 2$. As a comparison point, Progol's results reported in the previous chapter are also included. Appendix B contains the settings and mode declarations per dataset used for Mio and TILDE; for FOIL the defaults values were used.

As Figure 6.4 and Table 6.1 show, the classification error of MioM is lower than that of TILDE, FOIL and Progol, with mesh being an exception, where TILDE obtains the lowest error. According to a t-significance value of $\alpha = 0.005$, MioM's accuracy is statistically higher than that of FOIL in the chess, student loan,

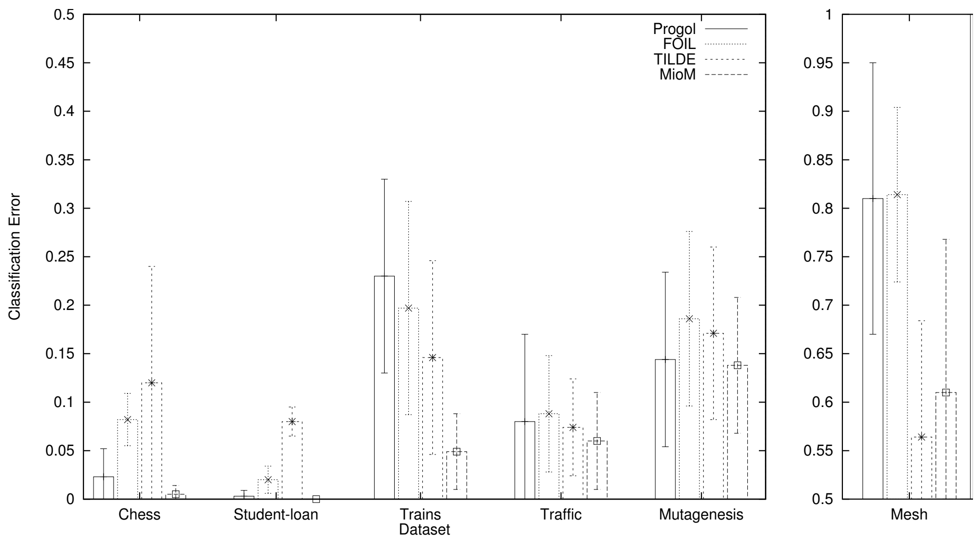


Figure 6.4: Classification error of Progol (left), FOIL (2nd bar), TILDE (3rd bar), and MioM (right) per dataset

Dataset	Progol	FOIL	TILDE	MioM
Chess	0.02	0.08	0.12	0.01
S.Loan	0.003	0.02	0.08	0.00
Trains	0.23	0.20	0.15	0.05
Traffic	0.08	0.09	0.07	0.06
Mutag.	0.14	0.19	0.17	0.14
Mesh	0.81	0.81	0.56	0.61

Table 6.1: Average error per system per dataset

trains, and mesh design domains. MioM's accuracy is statistically higher than that of TILDE in the chess, student loan, and trains datasets. Finally, MioM's accuracy is statistically higher than that of Progol in the trains, student loan, and mesh datasets. Thus, macros significantly improve accuracy compared with template-based lookahead (TILDE) and determinate literals (FOIL), and obtain an average accuracy which positively compares with that obtained by exhaustive search (Progol).

Table 6.2 shows the average run time on a Sun Blade 100 (500 MHz and 128 MB of RAM) of Progol, MioM, TILDE and FOIL per dataset. One can see that while macros represent a significant improvement in accuracy with respect to the other systems, they attain a middle place in terms of running time. There is reason to believe that MioM's longer running times are in part due to imple-

Dataset	Progol	MioM	TILDE	FOIL
Chess	1.2s	4.6s	1.1s	0.8s
S.Loan	17s	7.1s	1.6s	24.7s
Trains	54s	100s	0.2s	0.1s
Traffic	182s	113s	2.8s	1.7s
Mutag.	1h18m	48m	18.6s	2.2s
Mesh	13h55m	1h48m	35.3s	9.1s

Table 6.2: Average run time per system per dataset

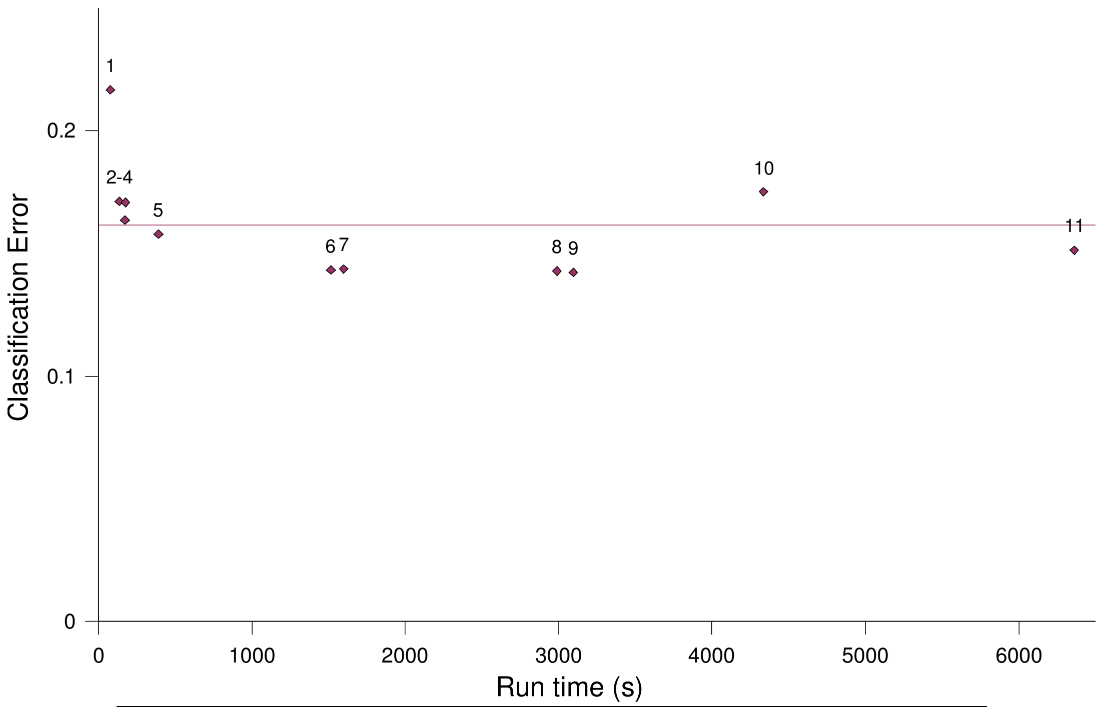
mentation issues: Mio's main components are implemented in Java and calls to SICStus Prolog are done using Jasper (a Java-SICStus interface); while FOIL is implemented in C and TILDE is implemented in ilProlog which is a built-in high performance Prolog system with special purpose features for ILP. To determine the impact of the implementation, we executed a single query to obtain the coverage of a four-body-literal clause (given identical examples and background knowledge) using TILDE and Mio. TILDE takes on average 0.01s to execute this query, while Mio takes 0.10s. Generating the macros is a minor time overhead since, on average, it takes 0.1 milliseconds to automatically construct a macro and 130 macros are generated per covering iteration (see Table 5.4 on page 59). Hence, macros are not the efficiency bottleneck of the system.

6.4.2 Comparing Macros with Fixed-depth Lookahead and Beam-Search

To compare macros with fixed-depth lookahead and beam-search in terms of classification error and run time, we ran MioM and MioL using Algorithm 6.1 with different values for the parameters b and s. MioM was run with s set to 1, 2, and 3, and $b = 1$. MioL was run with eight different parameter settings: using hill-climbing ($s = 1$ and $b = 1$); with 2, 3 and 4-step lookahead ($s = 2 \ldots 4$ and $b = 1$), and using beam-search with various beam widths ($s = 1$ and $b = 5, 20, 80, 160$).

Figure 6.5 (top) shows the average classification error across all six datasets obtained by each parameter setting. The horizontal line at 0.162 represents the average classification error of all settings reported. The table on the bottom of Figure 6.5 shows the values of the data points. For comparison, MioM using IDA* reports an average classification error of 0.149 across the same six datasets with an average run time of 4095 seconds.

In the experiments, hill-climbing (see data point 1 in Figure 6.5) has, as we expected, the highest classification error. The lowest classification error is obtained



Point	**Approach**		**Run Time**	**Error**
1	Hill-climbing	-	76s	0.217
2	Beam-search	$(b = 5)$	135s	0.171
3	Fixed-depth L-ahead	$(s = 2)$	171s	0.164
4	**MioM**	-	175s	0.171
5	Beam-search	$(b = 20)$	390s	0.158
6	Beam-search	$(b = 80)$	1514s	0.143
7	**MioM**	$(s = 2)$	1598s	0.144
8	Beam-search	$(b = 160)$	2990s	0.143
9	**MioM**	$(s = 3)$	3096s	0.142
10	Fixed-depth L-ahead	$(s = 3)$	4336s	0.175
11	Fixed-depth L-ahead	$(s = 4)$	6362s	0.151

Figure 6.5: Up: Average classification error and running times across all six datasets using macros, hill-climbing, fixed-depth lookahead and beam-search. Bottom: Reference table with the values of the data points shown in the diagram

using macros with lookahead (data points 7 and 9) and beam-search with the beam width set to 80 and 160 (data points 6 and 8). However, beam-search has the drawback that the beam width has to be tuned by trial-and-error. In our case, we tried in total six different beam width values for every application domain.

Increasing the amount of fixed-depth lookahead beyond two (data points 10 and 11) does not pay off because of the long running times and the marginal decrease

Approach	**Beam-search**	**Fixed-depth L.**	**TILDE**	**Progol**	**FOIL**	**Hill-climb.**
MioM ($s = 2$) vs.	0-0-6	2-0-4	3-0-3	4-0-2	5-0-1	5-0-1
Beam-search ($b = 80$) vs.		2-0-4	3-0-3	4-0-2	5-0-1	5-0-1
Fixed-depth L-ahead ($s = 2$) vs.			3-1-2	4-0-2	4-0-2	4-0-2
TILDE (template-based) vs.				1-2-3	1-2-3	2-0-4
Progol (exhaustive search) vs.					2-0-4	2-1-3
FOIL (determinate literals) vs.						2-1-3

Table 6.3: Win-loss-tie comparison between the approaches (row vs. column) in terms of the no. of domains with a significant ($\alpha = 0.05$) difference in accuracy

in classification error (which only occurs for data point 11). In addition, the behaviour of beam-search and macros is more stable than that of fixed-depth lookahead since there is no descent in accuracy (in fact, their accuracy seems to converge) and no oversearching [90] occurs when the beam width or the amount of lookahead is increased.

6.4.3 Results Summary

Table 6.3 shows a win-loss-tie comparison between all the approaches' accuracies. For this table, we select the best configuration found for fixed-depth lookahead ($s = 2$) and beam-search ($b = 80$). As shown in the first row of Table 6.3, MioM is more accurate than all the other approaches but beam-search, with 0.95% statistical significance.

The most effective approaches to reduce hill-climbing's myopia problem are macros with lookahead ($s = 2$) and beam-search with the beam width set to 80 clauses, which obtain practically the same accuracy on every domain and have similar run times (1598 and 1514 seconds respectively). In addition, macros, in contrast to some of the other approaches, can be computed fully automatically, do not require user involvement nor special domain properties such as determinacy, and their performance is less sensitive to domain-dependent tuning of parameters.

6.5 Related Work

Related approaches to the use of macro-operators to alleviate the myopia of hill-climbing search are FOIL's determinate literals [86], Relational clichés [98], TILDE's lookahead [5] and pathfinding [94].

- **Determinate literals**

  Determinate literals (Definition 3.8) appeared first in Golem [72] and were subsequently adopted by FOIL and LINUS [58]. This approach works as follows. In a first step, all determinate literals are added to the clause to be refined. In a second step, refinements containing consumers of these literals are evaluated, and the best one is selected. Finally, all determinate literals without a consumer are removed from the clause.

  Determinate literals are the subset of dependent providers which are uniquely satisfied by all the examples. That is, determinate literals must be uniquely satisfied by all the positive examples while a dependent provider must be satisfied by all the examples but can be multiply satisfied. Determinate literals' approach requires determinacy in the application domain. However, determinacy is not a property present in many application domains. For example, in the eastward trains domain, a train can have multiple cars, or in the mesh design domain, a node has multiple neighbours.

- **Template-based lookahead**

  Relational clichés and TILDE's lookahead refine a clause based on user-defined templates. Basically, in these two approaches, the user has to provide a template for every provider-consumer match. For example, in the mesh design domain the user has to combine the relations **neighbor** and **opposite** with every relation about edge attributes. This equates to 36 templates (see B.2.5). In macros, on the other hand, the user only has to identify the dependent providers; in fact, not even that since there is a procedure for automatically finding the dependent providers in a given domain as explained in Section 5.3.3.

  In addition, in both approaches, relational clichés and TILDE's lookahead, the system can, but does not have to, use the templates provided to refine a clause and by doing this the system may consider clauses which are not legal subsequences of literals.

  Therefore, a macro-based approach has two advantages over template-based approaches such as relational clichés and TILDE's lookahead: 1) macros do not imply extra work for the user, and 2) with macros the system considers only legal subsequences of literals.

- **Pathfinding**

  In relational pathfinding, a domain is viewed as a graph with constants as its nodes and the relations which hold between the constants as its edges.

The learning system is then restricted to learn clauses that correspond to a path through the graph; i.e., all the constants in the clause are in the same path formed by the relations which hold among them. Relational pathfinding improves the accuracy of a greedy learning system but, as pointed out by Richards and Mooney, is less efficient than determinate literals and relational clichés.

Pathfinding can be seen as restricting the macro-based refinement operator to add a macro to a clause only if there is a variable intersection between the macro and the body of the clause. Macros are more general than relational pathfinding because they allow the system to learn clauses formed by two or more disconnected subpaths in the graph.

6.6 Summary

In this chapter, we showed that macros are also suitable for alleviating the myopia of hill-climbing search. Myopia in hill-climbing mainly occurs because hill-climbing does not consider the existence of dependent providers and the inability of the evaluation function to deal properly with these kind of literals. Macros reduce hill-climbing's myopia problem because the evaluation function is only applied to legal subsequences of literals; i.e., to clauses whose quality can be more accurately assessed because they have discriminative power and can in fact be a solution.

Empirical results on several application domains showed that a greedy learner using macros exhibits significantly lower classification error than other systems using other techniques such as fixed-depth lookahead, template-based lookahead or determinate literals. What is more, macros are automatically computed, can be employed in any domain with dependent providers, and their performance does not rely on user-defined parameters or user-defined templates.

7 Improving the Stability of Example-driven Learning

This chapter deals with the instability of example-driven learning. In the example-driven approach, a learning system employs one or a few positive examples to guide the search for hypotheses. This approach is quite effective to learn complex rules (such as the rules usually needed in structural/topological domains); however, it has the drawback that since only one or a few individual examples are selected as basis for generalization in each round, the choice of examples can have a significant effect on the quality and contents of the learning results. Thus, the user can not rely on obtaining identical, or at least identically performing results in different runs with the same data; i.e., the learning system is *instable* (see Definition 7.1 below). What is more, as pointed out in [65], example-driven algorithms are easily misled by a few noisy examples and are hence less robust when the training data contains errors.

While it is well known that example-driven systems potentially exhibit the kind of instability described above, such stability issues have not been considered in great detail in the literature. In this chapter, we explore the stability of two example-driven multirelational learning systems, namely Progol and Mio. We also examine one possible solution to the problem, presenting an algorithm which relies on stochastically selected examples and parallel search. We implemented this algorithm in Mio and carried out experiments on four application domains. The empirical results obtained show that our algorithm almost eliminates the instability of example-driven search with limited additional effort. Our approach also delivers a numerical characterization of the degree of instability of a particular application domain, providing additional insight about the behavior of ILP problems.

In the next section, we formally define what we mean by stability. In section 7.2, we describe how we reduce the instability of example-driven learning by performing parallel search on stochastically selected examples. Section 7.3 presents our empirical results, and related work is discussed in Section 7.4.

7.1 Stability in Example-driven Learning

Example-driven learning consists in using generalizations of one or a few individual examples to constrain the search space and guide the search for appropriate hypotheses. The expectation is that, by using individual examples as starting points or seeds, the learning system starts the learning process closer to appropriate hypotheses than when no seed is used. How an example is generalized is a specific feature of each example-driven system. For example, Progol [68] and Mio use an example to derive a bottom clause using inverse entailment (Section 4.2); while Golem [72] obtains the *relative least general generalization* (rlgg) of two examples.

Since typically not every selected example e leads to equally good generalizations, the results obtained by example-driven learning systems can be affected by the order in which the examples are taken into consideration by the system. This lack of stability might have negative consequences such as increasing the likelihood of obtaining sub-optimal results (i.e., theories with lower prediction power), and getting different theories from the same training examples. For instance, Table 7.1 shows two theories about the diagonal movement of the queen in chess. Both theories were learned by Progol from exactly the same set of examples; the only difference was the order in which the examples were given to the system. In this case, the learning system is instable. We therefore define stability after renaming variables (Definition 2.26) as follows.

Definition 7.1 – Stability. *A learning system is defined as* stable *(against reordering) when, given reordered but otherwise identical training examples E (Definition 3.1),*

(a) *it obtains identical sets of clauses:* syntactic stability*, or*
(b) *if all different results have equal prediction power:* predictive stability*.*

This stability definition directs toward two possible measures of stability, or rather, instability: *accuracy distance* and *syntactic distance*. We designed our own instability measures, because no standard measures exist to compare the stability of rule learning systems.

Definition 7.2 – Accuracy Distance. *Let A and B be two theories obtained on reordered example sets, and let $\alpha_A \in [0,1]$ and $\alpha_B \in [0,1]$ be the classification*

Theory A	Theory B

`move(queen,pos(A,B),pos(C,D)):-`
`    diff(B,D,5), diff(A,C,5).`
`move(queen,pos(A,B),pos(C,D)):-`
`    diff(B,D,4), diff(A,C,4).`
`move(queen,pos(A,B),pos(C,D)):-`
`    diff(B,D,3), diff(A,C,3).`
`move(queen,pos(A,B),pos(C,D)):-`
`    diff(B,D,2), diff(A,C,2).`
`move(queen,pos(A,B),pos(C,D)):-`
`    diff(B,D,1), diff(A,C,1).`
	`move(queen,pos(A,B),pos(C,D)):-`
	`    diff(B,D,E), diff(A,C,E).`

Table 7.1: Instability example: theory A and theory B were obtained by the same ILP system from the same training examples

error (Definition 3.2) of A and B. The accuracy distance *between A and B is defined as follows.*

$$\delta_{acc}(A, B) := |\alpha_A - \alpha_B|. \tag{7.1}$$

For example, assume Theory A in Table 7.1 has a classification error of 0.05 and Theory B of 0.0, then the $\delta_{acc}(A, B) = |0.05 - 0| = 0.05$.

Definition 7.3 – Syntactic Distance. *Consider A and B as sets of clauses, and $A_{bag} := bagOf(A \setminus (A \cap B))$ and $B_{bag} := bagOf(B \setminus (A \cap B))$ as bags of literals. The* syntactic distance *between A and B is calculated as follows.*

$$\delta_{syn}(A, B) = \begin{cases} \frac{|A_{bag} \setminus B_{bag}| + |B_{bag} \setminus A_{bag}|}{|A_{bag}| + |B_{bag}|} & \text{if} A \cap B \neq \emptyset, \\ 1 & \text{if} A \cap B = \emptyset \end{cases} \tag{7.2}$$

where $\mid S \mid$ is the cardinality of S.

Note that A_{bag} and B_{bag} contain the clauses' literals as single elements and may have duplicate literals. Let us consider again the theories shown in Table 7.1. In this case, $A \cap B = \emptyset$ (i.e., there is no clause in common between both theories) and thus $\delta_{syn}(A, B) = 1$.

We define the degree of stability of a learning system on a particular problem as follows.

Definition 7.4 – Instability Measures. *Let E be training examples from an application domain, L an example-driven learning system, and $T_1, \ldots, T_n$ be the theories obtained by running L on n random permutations of E. The* instability *of L on E can be measured as follows.*

- *The* syntactic instability *of L on E (based on n permutations) is calculated with the following equation.*

$$\frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \delta_{syn}(T_i, T_j)}{n}, \tag{7.3}$$

i.e., the average pairwise syntactic distance between the T_i.

- *The* predictive instability *of L on E (based on n permutations) is given by the following equation.*

$$\frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \delta_{acc}(T_i, T_j)}{n}, \tag{7.4}$$

i.e., the average pairwise accuracy distance between the T_i.

Note that a system having zero syntactic instability has as well zero value in the predictive stability measure.

7.2 Enhancing the Stability of Example-Driven Learning

To increase the stability of example-driven learning, we propose parallel stochastic search on several examples because with parallel search generalization is based on more than one example. Basically, we integrate parallel search with the covering algorithm, using previous solutions both to increase quality and to avoid unnecessary search through iterative deepening. This algorithm was implemented in Mio.

The parallel search is performed as follows. At each iteration, Mio takes randomly a subset of positive examples of the same target predicate and creates an independent search agent[1] for each example in the subset to search its hypothesis space $H^{\perp}$ (Definition 4.5) as shown in Figure 7.1. Each agent searches for a solution (Definition 4.8) using the search strategy selected by the user. This search strategy can either be IDA*, hill-climbing or beam-search.

When IDA* is performed, each agent keeps a *transposition table*[2] during the search. The transposition table stores the clauses which cannot be improved by adding a new literal, and is used to prune the search tree in subsequent IDA*

[1]An agent can be seen as an independent process or a thread.

[2]Transposition table is a term used in game-playing that refers to a search enhancement. Basically, a transposition table is a large cache in which newly expanded states are stored.

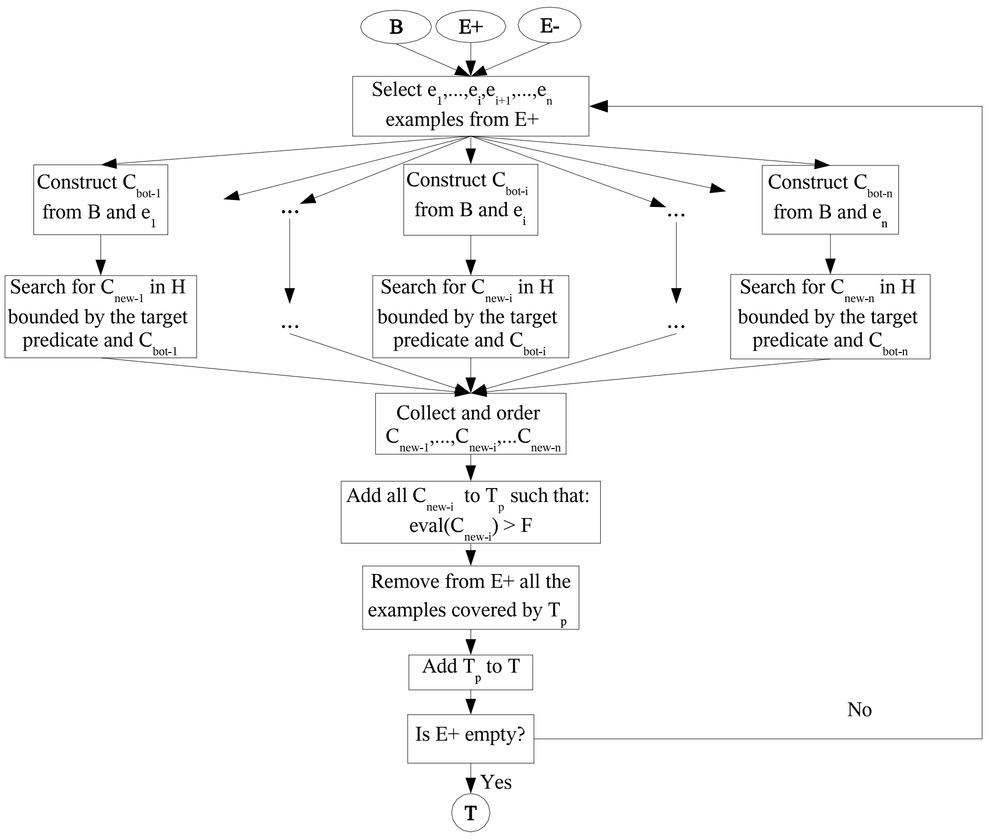

B
E+
E-
Select $e_1,...,e_i,e_{i+1},...,e_n$ examples from E+
Construct C_{bot-1} from B and e_1
...
Construct C_{bot-i} from B and e_i
...
Construct C_{bot-n} from B and e_n
Search for C_{new-1} in H bounded by the target predicate and C_{bot-1}
...
Search for C_{new-i} in H bounded by the target predicate and C_{bot-i}
...
Search for C_{new-n} in H bounded by the target predicate and C_{bot-n}
Collect and order $C_{new-1},...,C_{new-i},...C_{new-n}$
Add all C_{new-i} to T_p such that: eval(C_{new-i}) > F
Remove from E+ all the examples covered by T_p
Add T_p to T
Is E+ empty?
No
Yes
T

Figure 7.1: Parallel search algorithm

iterations. In addition, after finishing searching for a given clause-length (i.e., an IDA* iteration is over), the agents check a global best evaluation value (GBE-value) to stop searching when the GBE-value cannot be outperformed. This is only done for IDA* because the deepening mechanism of IDA* allows to stop and restart the search without problem. In the case of hill-climbing and beam-search, the search is completed without interruption.

After every agent stops, all the agents' solutions are placed in a descending ordered list according to an evaluation function (Definition 3.12). Mio traverses this list and adds to the final theory T the solutions whose evaluation value is above a user-defined threshold F and cover enough positive and few enough negative examples in E. Every time a solution is added to T, all the positive examples covered by it are removed and the remaining agents' solutions are reordered. Those examples which at the end do not provide a solution are added to a losers list. If an example in the losers list is selected again in a subset, no search is performed and its value is the initial value of the GBE-value.

The number n of positive examples taken in each iteration is determined by three user-defined parameters. One parameter indicates which fraction of the remaining examples to be covered has to be taken, the other is the minimum number of examples that Mio takes, and the third one indicates the maximum number of examples that Mio takes. The percentage of remaining examples taken for parallelization can be seen as another measure for the instability of a domain: the more examples necessary to obtain stability, the less stable a domain is. We thus can measure stability of an application domain in a third way by determining the minimum percentage of examples required to achieve perfect stability (i.e., $\forall A, B \quad \delta_{syn}(A, B) = 0$).

Definition 7.5 – Parallelism for Perfect Stability. *Let E be training examples from an application domain, and L an example-driven learning system. Let $s \in [0, 1]$ be a parameter controlling the amount of parallel search performed by L. The* required parallelism *of L on E is the smallest number p such that the syntactic instability of L on E is zero when using p as a value for s.*

7.3 Empirical Results

To explore the use of parallel stochastic search as a solution against the instability of example-driven, we carried out experiments on four application domains, namely, chess moves, student loan, eastward trains and mesh design which are described in Chapter 5 (page 55). The goals of our experiments were:

1. to determine in which degree two example-driven systems (Progol and Mio) are affected by instability;
2. to find out whether the search strategy employed has an influence on the stability level;
3. to explore the suitability of stochastic example selection and parallel search as a solution to the instability problem, and
4. to find out whether the required parallelism for perfect stability varies according to the application domain.

We decided to use Progol (Section 3.3.1) because it is the most commonly used example-driven system in ILP publications of the last 7 years, and, the same as Mio, constructs the bottom clause by inverse entailment. Mio differs from Progol in several aspects such as including parallel search, hill-climbing search and beam-search; enforcing type strictness (Definition 4.3); selecting stochastically

Dataset	Avg. δ_{syn}			Avg. δ_{acc}		
	Progol	Mio		Progol	Mio	
		IDA*	HC		IDA*	HC
Chess moves	0.15	0.02	0.02	0.03	0.01	0.01
Student loan	0.23	0.17	0.17	0.01	0.01	0.01
Eastward trains	0.07	0.16	0.06	0.03	0.03	0.00
Mesh Design	0.03	0.09	0.10	0.00	0.00	0.01

Table 7.2: Syntactic and predictive instability of Progol and Mio on four application domains based on five random permutations

the examples to guide the search; and supporting the use of macros and active inductive learning (described in Section 8.2).

For the mesh design dataset, the examples corresponding to one structure were set as testing data and the examples of the nine other structures were arranged in five different random permutations. For the three other datasets, 20% of the examples available was set apart as testing data and the training examples were arranged in five different random permutations. The systems were given these five reordered but otherwise identical training examples, and the theories obtained were compared using the stability measures defined in Section 7.1. Mio was run twice on each permutation: one using IDA* and one using hill-climbing with 2-step lookahead. In these experiments Mio used the macro-based refinement operator ρ_M (Definition 5.9).

Table 7.2 shows the syntactic and predictive instability of Progol and Mio. The instability values were obtained with Equations 7.3 and 7.4 respectively. Mio using hill-climbing seems to be on average the least affected by instability.

Once the instability levels of Progol and Mio per dataset were determined, the amount of parallelism (i.e., percentage of positive examples taken for parallelization) performed by Mio was gradually increased to try to reach perfect stability on each application domain. The percentage of positive examples required for perfect stability indicates the degree of instability intrinsic to every application domain (see Definition 7.5). Table 7.3 shows the minimum percentage of examples required to achieve perfect stability per application domain and the run-time penalty. Perfect stability was achieved on all datasets but mesh design. Run-time penalty is the factor in which the run-time of Mio increases compared with a version of Mio taking one example for iteration. We have an increase in run-time because the parallelization is actually done sequentially in a one-processor computer. All the experiments were done in a Sun Blade 100 (500 MHz and 128 MB of RAM).

Dataset	Parallelism for Perfect Stability		Run Time seconds		Run Time Penalty	
	IDA*	HC	IDA*	HC	IDA*	HC
Chess moves	5%	5%	5.8	5.4	1.1	1.2
Student loan	50%	50%	208	149	22.1	17.1
Eastward trains	40%	75%	284	684	3.7	7.7
Mesh Design^{+}	N/A	70%	N/A	11800	N/A	3.4

$^{+}$ Only predictive stability was reached.

Table 7.3: Percentage of positive examples per application domain taken by Mio for parallelization to obtain perfect stability, and Mio's run time by performing parallel search

Dataset	Accuracy without Parallel Search (%)			Accuracy with Parallel Search (%)	
	Progol	Mio		Mio	
		IDA*	HC	IDA*	HC
Chess moves	97 ± 2.5	99.5 ± 0.9	99.5 ± 0.9	100 ± 0	100 ± 0
Student loan	99 ± 0.7	99.6 ± 0.8	99.6 ± 0.8	100 ± 0	100 ± 0
Eastward trains	57 ± 3.3	85.0 ± 3.3	91.7 ± 0.0	83.3 ± 0	83.3 ± 0
Mesh Design	36 ± 0.0	78.6 ± 0.0	77.9 ± 1.4	N/A	78.6 ± 0

Table 7.4: Average accuracy obtained by the systems per dataset

Table 7.4 shows the average accuracy and standard deviation per dataset obtained by Progol and Mio without parallel search, and by Mio with parallel search. In the trains dataset, loss in accuracy is observed by performing parallel search specially when using hill-climbing search. An explanation could be that hill-climbing search has null predictive instability without parallel search and by performing parallel search to obtain perfect stability oversearching [90] occurs. The same phenomenon occurs in the mesh design domain with IDA*; that is, Mio with IDA* has null predictive instability without parallel search and by increasing the amount of parallelism up to 60% we observed that a decrease in accuracy occurs. In mesh design, syntactic stability was not reached.

With our empirical results, we have seen that two example-driven systems are indeed affected by re-ordering of the training examples; that parallel search based on randomly selected examples reduces the effects caused by reordering of the examples; and, that the minimum amount of parallelization required seems to be related to some intrinsic features of each dataset. However, parallel search is not yet a perfect solution against the instability of example-driven systems;

specially in domains with numerical data. For domains with numerical data, an alternative could be to combine discretization with parallel search. In addition, the impact of oversearching has to be further investigated. It might be that striving syntactic stability is too ambitious and predictive stability is enough for most purposes.

7.4 Related Work

Work related to the one presented in this chapter is the use of parallelization to speed up the learning process in ILP systems.

Matsui et al. [63] examine three parallelization approaches to speed up FOIL: partition of the search space, partition of the training examples, and partition of the background knowledge. With the last two alternatives, the authors reported a 4-fold speed-up of FOIL using five processors. However, by increasing the number of processors beyond five no speed-up of the system is obtained because of communication overhead. Similarly, Fujita et al. work on a parallel implementation of Progol. In [38], the design and a partial implementation of parallel-Progol is discussed, but no empirical results are provided.

Dehaspe and De Raedt [22] created a parallel version of CLAUDIEN [19] based on recursively partitioning the search space and processing these partitions concurrently to speedup the learning process. However, since in this approach multiple processors participate in the same search (instead of creating independent searches as we do), it could not be applied to increase the stability of example-driven learning because the learning results would still be based on one single example.

Graham et al. present in [43] a parallel inductive logic search, which is implemented to run on special hardware (an eight node Beowulf cluster), to expedite the drug design cycle. Initial results reported by these authors show an almost linear speed-up of the learning process. However, similar to the work done by Dehaspe and De Raedt, this approach is unsuitable for increasing the stability of example-driven learning because a master processor distributes the work load from the same search to other processors; i.e., one single example is still taken as basis for generalization.

In addition, contrary to us, all the authors previously mentioned do not consider parallel search as a solution to the instability problem of example-driven learning, and, consequently, do not explore the stability of their results.

Using several stochastically selected examples to guide the search in example-driven learning was first done in Golem [72]. Golem takes a user-defined number of positive examples at each iteration of the covering algorithm, computes their rlggs, and selects the one rlgg with the greatest coverage. However, contrary to our approach, no parallelization is added to the covering algorithm.

An interesting work in exploiting the instability of example-driven learning to create ensembles[3] is done by [27]. Basically, in [27] the authors compare the performance of ensembles created by using the theories obtained by an example-driven learning system, which is run on identical but re-ordered training examples, with the performance obtained by using bagging. However, an ensemble is more complex than a single theory and thus harder to understand.

7.5 Summary

In this chapter, we study the stability of example-driven learning against reordering of the examples. For that, the concept of stability against reordering is formally defined and two reasonable stability measures are suggested. In addition, parallel stochastic search is proposed as a solution to provide stability against reordering of the training data. This approach is explored in Mio.

Experimental results in four datasets show that two example-driven multirelational learning systems are affected by re-ordering of the examples, and that stochastic selection of examples and parallel search are indeed a viable solution to provide stability against reordering of the training data.

[3]An ensemble is a classifier which combines the predictions of various classifiers into a single prediction [24]

8 Learning Minesweeper with ILP

This chapter describes how the techniques presented in previous chapters together with active inductive learning, which is introduced in this chapter, make possible for Mio to discover a Minesweeper playing strategy. We focus on learning playing strategies for Minesweeper because Minesweeper is a highly structured application domain, and, thus, an ideal test-bed for the work presented in this thesis. In fact, the task of learning rules to deduce Minesweeper moves proved itself to be an arduous test for current general purpose multirelational learning systems such as FOIL, Progol and TILDE. Experimental results obtained by playing Minesweeper using Mio's playing strategy show a better performance than that obtained on average by non-expert human players.

Given the difficulty of hand-crafting playing strategies for game playing programs, AI researchers have always been interested in the possibility of automatically learning such strategies from experience. However, with the exception of TD-Gammon which uses reinforcement learning [101] and LOGISTELLO which applies GLEM [9], most of the playing strategies and heuristics used in game playing programs are coded and tuned per hand instead of automatically learned. Games are one of the oldest domains used in AI because they provide a controllable environment to try out new techniques and study problems found in real-world domains. Multirelational learning is a good option for learning game playing strategies because many games involve structural and topological components which are difficult to describe using a propositional representation.

In the next section, we describe Minesweeper, discuss its complexity and define it as a multirelational learning task. In Section 8.2, we explain active inductive learning. Section 8.3 describes the background knowledge used. Section 8.4 shows our empirical results on the effectiveness of the learning techniques, the strategy obtained and its performance at game playing. Related work is surveyed in Section 8.5 and Section 8.6 concludes.

8.1 Minesweeper

Minesweeper is a popular one–player computer game written by Robert Donner and Curt Johnson which was included in Microsoft Windows© in 1991. At the beginning of the game, the player is presented with a $p \times q$ board containing pq tiles or squares which are all blank. Hidden among the tiles are M mines distributed uniformly at random on the board. The task of the player is to uncover all the tiles which do not contain a mine. At each turn the player can select one of three actions (moves): to mark a tile as a mine; to unmark a tile; and to uncover a tile. In the last action, if the tile contains a mine, the player loses; otherwise, the number of mines around the tile is displayed. For example, in the 4×4 board depicted in Figure 8.1 center, the number 2 located on the second row from top indicates that there are exactly two mines hidden among the eight blank neighbouring tiles.

Figure 8.1 shows two possible starting sequences of a Minesweeper game. In the top right board, the player uncovers a tile which has zero mines around and several other tiles are automatically revealed. Most Minesweeper implementations automatically unveil all the neighboring tiles of a square with zero mines around when it is uncovered, which actually does not increase the player's chances of winning the game. In the bottom right board, the player steps in a mine and loses the game.

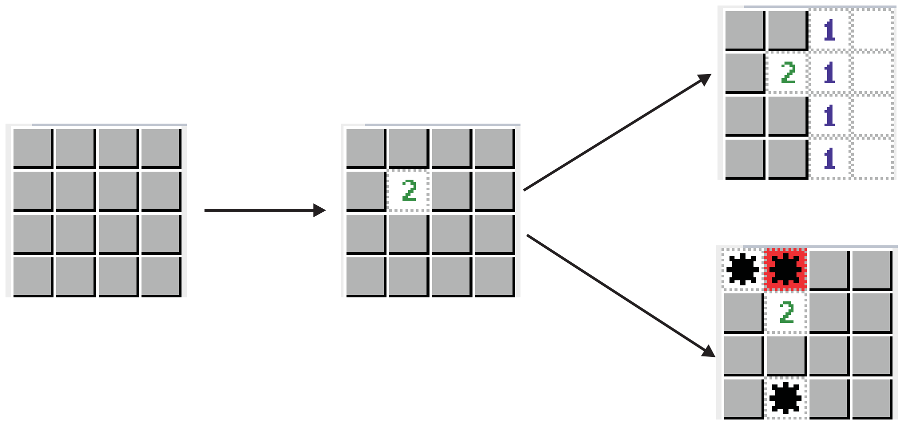


Figure 8.1: Two possible starting sequences of a Minesweeper game. Left: Starting position where all squares are blank. Center: Uncovering a tile - the first move is always a guess. Right: Two possible outcomes for the second move.

Although the simplicity of its rules makes Minesweeper look deceptively easy, playing the game well is indeed challenging: A player requires logic and arithmetic reasoning to perform certain moves given the board state, and probabilistic reasoning to minimize the risk of uncovering a mine when a safe move cannot be done. Playing well Minesweeper means that the player never makes a risky move (i.e., a guess) when there is some blank tile which can be uncovered safely.

8.1.1 Why Is Minesweeper Interesting?

Minesweeper has been shown to be NP-complete by simulating boolean circuits as Minesweeper positions [49]. Kaye describes the *Minesweeper consistency problem* as the problem of determining if there is some pattern of mines in the blank squares that give rise to the numbers seen in a given board partially filled with numbers and marked mines, and thus determining that the data given is consistent.

One realizes the complexity of the game by calculating an estimate for the size of its search space. Consider an 8×8 board with $M = 10$ mines; in this case at the beginning of the game the player has $pq = 64$ tiles from which to choose a move (i.e., a tile to uncover) and in the last move, assuming the player does not uncover a mine, there are 11 tiles from which to choose one. This leads to $54! \approx 10^{71}$ possible move sequences to win a game. Alternatively, one can calculate the probability of a random player winning a game. In the first move the probability that the random player chooses a tile which does not contain a mine is 54/64, and in the last move it has 1/11 chance to choose the only tile without a mine. Then, the probability of a random player winning a game is

$$\frac{1}{\binom{64}{54}} \approx 10^{-12}$$

and that is only for the easiest playing level!

Another measure of the complexity of Minesweeper is the number of games won on average by non-expert human players. To estimate the average human performance playing Minesweeper, we carried out an informal study. In the study, eleven persons who have played Minesweeper before were asked to play at least ten times in an 8×8 board with 10 mines. Every participant was told to aim for accuracy rather than for speed, which is different from the way people usually play Minesweeper. Another difference with most Minesweeper implementations is that in this study a player can hit a mine in the first move. In the study, a person won on average 35% of the games with a standard deviation of 8%.

8.1.2 Minesweeper as a Learning Task for ILP

In Minesweeper there are situations that can be "solved" with nontrivial reasoning. For example, consider Figure 8.2 left where the only available information about the board state are the numbers. After careful analysis one finds that the squares with an s (see Figure 8.2 right) do not contain a mine, the square with an m is a mine, and the state of the blank tiles cannot be determined if we do not know how many mines are hidden in the board. If we know how many mines are hidden in the board (either two or three) the state of all blank squares can be determined.

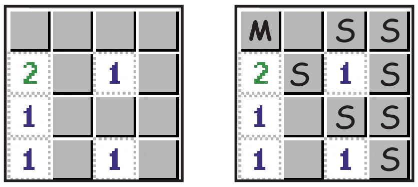


Figure 8.2: Left: Available information on a board. Right: Seven tiles can be determined safe (s) and one a mine(m).

There are other Minesweeper situations where the available information is not enough to identify a safe square or a mine, as in Figure 8.3, and the best option available to the player may be to make an informed guess, i.e., a guess that minimizes the risk of blowing up by uncovering a mine. However, in the worst case, as in Figure 8.3, the probability of containing a mine is the same for all the remaining blank squares and the player has to be fortunate to win.

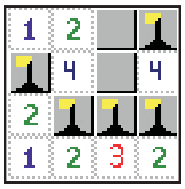


Figure 8.3: A Minesweeper board where the position of the last mine cannot be determined

In this work, we consider the learning task in Minesweeper to be the induction of a theory to identify all the *safe squares*[1] and squares with a mine which can be deduced given a board state. For instance, we want the system to learn clauses to

[1]A safe square is a blank tile which given the current board state cannot contain a mine.

classify all the blank tiles in Figure 8.2 either as *safe* or *mine*. The set of clauses found by the learning system is then used as a playing strategy of a Minesweeper program.

8.2 Actively Exploring the Instance-Space

By using Minesweeper as an application domain, we realized that games introduce an extra challenge for multirelational learning: The amount of examples (i.e., thousands) required to obtain information about most of the possible game situations. Considering thousands of examples when evaluating a clause slows down the learning process, because it has to be determined for every example in the training data whether or not it is covered by the clause. Since in games there are rare or exceptional cases which have to be covered by the induced playing strategy, random sampling is not an optimal solution because it may result that no instances of some minority case are present in the training data. This could prevent the system from learning a complete theory. Besides, as pointed out by Holte et al. [44], clauses or small disjuncts induced to cover these rare situations are more error prone than large disjuncts (i.e., clauses which cover a large number of training examples). To improve the efficiency of the exploration of the instance space and the quality of the small disjuncts learned, active learning [15] is included in Mio.

We refer to the combination of multirelational learning and active learning as *active inductive learning*. Active inductive learning consists of the following steps. At the beginning, Mio learns from few randomly drawn examples and when it has learned some clauses gives these clauses to an active learning server. The active learning server returns to Mio *counterexamples*. A *counterexample* is a positive example not covered by a theory T or a negative example covered by T. These counterexamples are selected from examples given by a random example generator (or random sampler). While Mio iterates on the new examples received, the server tests the clauses obtained against randomly drawn examples, discards all the clauses below a user-defined accuracy value, and collects new counterexamples. This validation step on the server side avoids overfitting[5] and improves the quality of the small disjuncts learned. These steps are repeated until a user-defined maximum number of iterations is reached or no counterexample is found.

Figure 8.4 depicts our active inductive learning framework, which is implemented in a client-server architecture. In this framework, the learner receives as input

[5]Overfitting refers to obtaining a theory with high classification error over new data despite null or almost null training error.

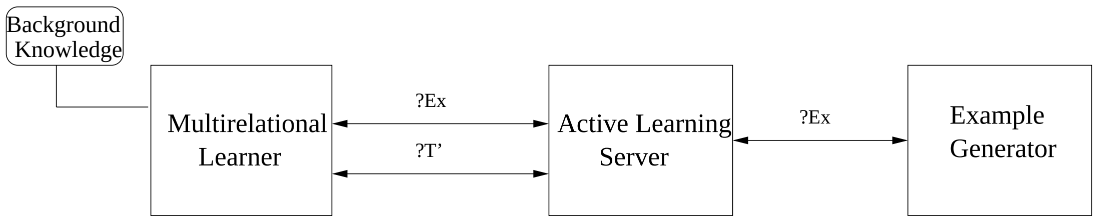


Figure 8.4: Active inductive learning framework

background knowledge B (and in the case of Mio a set of mode declarations), and the active learning server, similar to the minimally adequate Teacher in [2], is assumed to answer correctly the following two types of questions from the inductive learner.

1. ?**T**′.- The active learning server receives a set of Horn clauses T' and answers yes if no counterexample is found; otherwise, answers with a set of counterexamples.
2. ?**Ex**.- The active learning server acts as a random sampling oracle and selects examples from the whole domain according to a distribution P over all the elements in the domain and returns a set of correctly classified examples.

In our active inductive learning framework, the example generator is the domain-dependent component either by actually producing the examples or by accessing the file or database which contains the training data. In the case of Minesweeper, the example generator randomly generates Minesweeper board configurations and takes all blank tiles with at least two known (uncovered) neighbours as positive or negative examples. We restrict the examples to be those tiles with at least two known neighbours to guarantee that the board contains information about their state (safe or mine).

Active inductive learning is similar in spirit to integrative windowing [39] with two main differences: in our approach random sampling is done dynamically and a client-server architecture is used which allows to treat testing and learning as separate processes.

8.3 Background Knowledge

The background knowledge predicates provided to the learning systems about Minesweeper are shown in Table 8.1. In this table, predicates are described in the form of mode declarations. That is, a predicate **p** is defined by

1. **zoneOfInterest(+TU, +Board, -Zone)** .... returns in Zone the tiles which are determined neighbours of TU and the determined tiles which share an undetermined neighbour with them (see Figure 8.6 center on page 95).
2. **totalMinesLeft(+Board,-Int)** .... returns how many mines remained to be marked.
3. **allMinesInFringe(+Board, -Set)** .... gives the set of tiles in the *fringe* where all the remaining mines are. *Fringe* refers to all the blank tiles with a determined neighbour
4. **setHasXMines(-TD, +Board, +Zone, -Set)** .... gives in Set the undetermined neighbours of TD (TD is in Zone), and the number of mines hidden among them (see Figure 8.6 right).
5. **diffSetHasXMines(+Set1, +Set2, -Set)** .... returns in Set all and only the tiles of Set1 which are not also in Set2 and the number of mines hidden among the tiles in Set.
6. **notInSet(+TU, +Board, +Set)** .... is true when TU is not in Set.
7. **inSet(+TU, +Set)** .... is true when TU is a member of Set.
8. **lengthSet(+Set,+Int)** .... is true when Set contains Int tiles.
9. **minesInSet(+Set,-#Int)** .... returns the number of mines hidden among the tiles in Set.

Table 8.1: Minesweeper background knowledge

p({+/-/#}arg$_1$,...,{+/-/#}arg$_n$)
where **arg**$_i$ indicates the type of the ith argument and +, - or # indicates whether it is an input, output or constant argument. In the predicates listed in Table 8.1, the following types are used:

- **TD** is the ID of a determined or uncovered tile, i.e., a number $0 \dots 8$ is shown on the tile;
- **TU** is the ID of an undetermined or blank tile;
- **Board** is a board state description given as a list of $p \times q$ characters $0 \dots 8, m, u$;
- **Zone** is a list of determined tiles;
- **Set** is a composite type formed by an ordered set of undetermined tiles together with the number of mines hidden among those tiles; e.g., set([0,1,5], 1), and
- **Int** is an integer number.

The predicates in Table 8.1 were defined by abstracting the concepts used by humans when explaining their own Minesweeper playing strategies. These concepts were obtained from our own Minesweeper playing experience and from Minesweeper pages on the web. In addition, the examples are ground facts of the form **safe(TU,Board)** or **mine(TU,Board)**.

When using Mio's macro-based refinement operator ρ_M (Definition 5.9), literals having the predicates **zoneOfInterest**, **totalMinesLeft**, **allMinesInFringe**, **minesInSet**, **diffSetHasXMines** and **setHasXMines** are considered dependent providers (see Definition 5.1).

8.4 Empirical Evaluation

Experiments were carried out to determine the effects on the theory obtained and on the system's efficiency, of macros, the search strategy, and active inductive learning. To produce the training examples, we randomly generate board configurations and take all blank tiles with at least two determined neighbours as examples. If the blank tile does not contain a mine it is labeled as safe, otherwise it is labeled as mine. Afterwards, contradictory examples are removed. In the experiments, the learning task was to learn rules to identify safe tiles. These experiments differ from those presented in [82] in that the maximal clause length was set to nine literals instead of eight.

All the experiments with active inductive learning were performed with the same seeds which means that the same training examples are generated by the random sampler and that Mio selects the same examples to guide the search. In these experiments, Mio performed three active inductive learning iterations. For the experiments without active learning, we took five random samples from the examples used in the active inductive learning experiments. The size of the sample is equal to the number of examples received by the inductive learner (Mio) when performing active inductive learning (i.e., 40 positive and 34 negative examples). We carried out an extra experiment where Mio was given the complete set of examples (2890 positive and 1306 negative) used by the active learning server to test Mio's rules and select counterexamples; however, this experiment was stopped after Mio ran for 10 days. To reduce the running time of the experiments, we set the maximum number of clauses explored per search to 4000 clauses.

Mio Setting	Ave. No. clauses per search	% searches reaching 4000-clause limit	Ave. max. search depth	Ave.% of games won	Ave. run-time
IDA*	3283	72%	7.3	$43 \pm 10.8\%$	61h
IDA*+M	1440	22%	8.2	$35 \pm 9.5\%$	63h
IDA*+AL	2600	59%	7.3	51%	74h
IDA*+M+AL	1052	8%	8.0	52%	156h
HC	63	0%	8.4	$34 \pm 16.8\%$	10h
HC+M	68	0%	8.6	$28 \pm 13.5\%$	4h
HC+AL	60	0%	8.5	45%	19h
HC+M+AL	61	0%	8.7	51%	6h
HC2+M+AL	279	0%	7.9	52%	42h

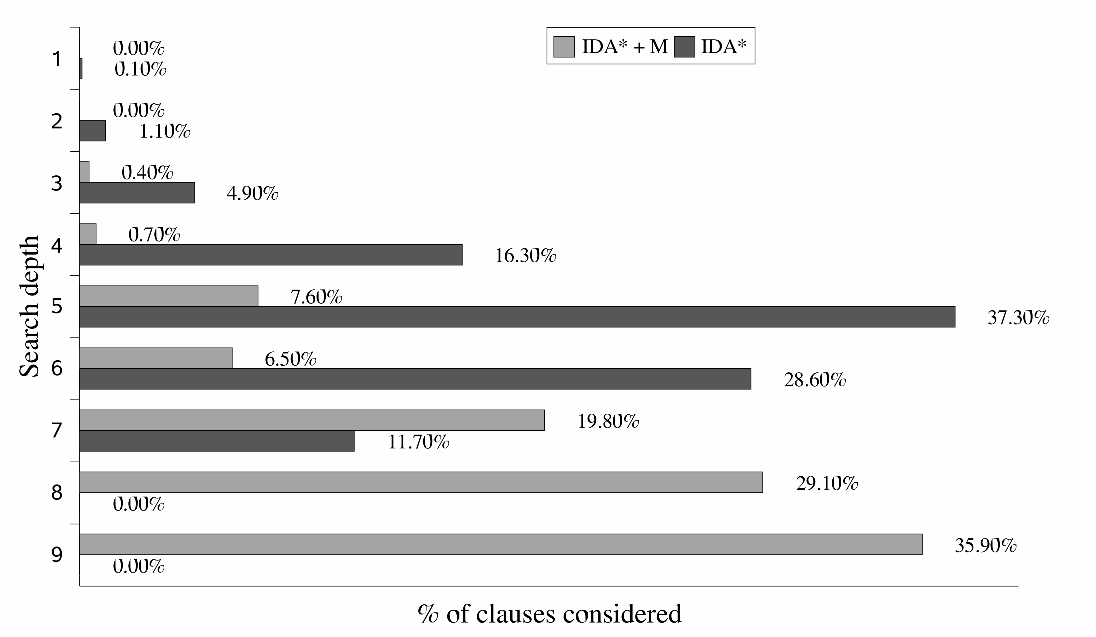
Table 8.2: Performance of various Mio settings used to learn rules about safe tiles (AL = Active Learning, HC = Hill-climbing search, HC2 = Hill-climbing search with 2-step lookahead, IDA* = Iterative Deepening A*, M = Macros)

8.4.1 Results

Table 8.2 shows the empirical results with various Mio settings. Active inductive learning, independently of the search strategy, avoids overfitting, discards overly general clauses and provides the system with the negative and positive examples required to learn more accurate clauses.

Macros in IDA* (see Chapter 5) reduce the search space, decrease the number of times the search reached the 4000-clause limit, and allow Mio to search deeper in the lattice. IDA*+AL learns Rules S-1 and S-2 in Table 8.3 but is unable to learn Rule S-3 because its average maximum search depth is 7.3 and Rule S-3 is at depth 8. With the macro-based approach (IDA*+M+AL) Mio finds Rule S-3.

Hill-climbing search without macros learns clauses with unneeded body literals. In some cases these extra literals do not have an effect on the coverage of the clause, e.g., HC+AL learns Rule S-1 with the extra literal **totalMinesLeft(BOARD,INT)**; but in other cases the unneeded literals reduce the coverage of the clauses learned. Macros reduce the myopia of hill-climbing search (Chapter 6) and avoid unneeded literals to be added to the clauses. HC+M+AL learns the same rules as IDA*+AL (Rules S-1 and S-2) but 10 times faster. Adding 2-step lookahead to hill-climbing (HC2+M+AL) allows Mio to learn Rule S-4 too.



Search depth (No. body literals)	1	2	3	4	5	6	7	8	9
Ave. evaluation time per clause (s)	0.3	0.5	0.6	0.7	1.5	4.0	6.6	10.6	28

Figure 8.5: Distribution of clauses considered by Mio during an active learning iteration according to their size. In the diagram, top bars correspond to IDA*+AL+M; bottom bars to IDA*+AL. The table in the bottom shows the average evaluation time per clause according to its size.

In Table 8.2, one observes that there is an increase in run-time when macros are used with IDA*. This seems contra-intuitive since in the macro-based approach (MioM) less clauses are considered per search and less searches reached the 4000-clause limit than in the literal-based version (MioL). To explain this behaviour we analyzed, during an active learning iteration, the clause distribution according to the clauses' size and measured how many seconds it takes to determine the coverage and consistency of each clause.

Figure 8.5 shows the percentage of clauses explored by IDA*+M+AL and IDA*+AL at each search depth. In this diagram, we observe that more than 80% of the clauses considered by the literal-based version contain less than seven body literals, while 80% of the clauses evaluated by the macro-based approach contain more than six literals. Due to this distribution of the clauses in the search space, the average evaluation time per clause of IDA*+AL is 2.6s, while IDA*+AL+M takes on average 14.8s to evaluate a clause. Thus, since the search space is re-

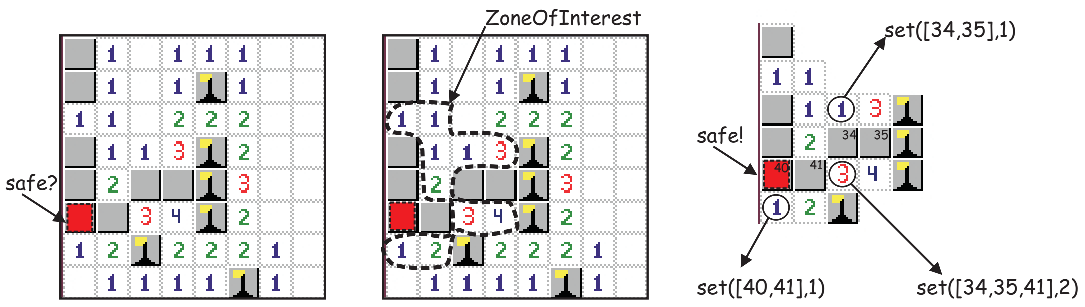


Figure 8.6: A clause to play Minesweeper learned by Mio. Left: Is the highlighted tile safe? Center: zoneOfInterest corresponding to the highlighted tile. Right: Applying difference operations to the sets determined by the tiles inside a circle is concluded that the tile 40 is safe.

stricted at 4000 clauses and MioL considers non-admissible clauses too, it ends evaluating more clauses with few body literals than MioM, and, since shorter clauses are evaluated faster than longer ones, MioM's run time is penalized by searching deeper in the refinement graph.

8.4.2 Theory Learned

Table 8.3 shows the theory with the highest winning rate which was obtained by joining the theories obtained by IDA*+M+AL and HC2+M+AL. Both settings learn Rules S-1 and S-2, and IDA*+M+AL learns Rule S-3 while HC2+M+AL learns Rule S-4. This combined theory has a winning rate of 53.6%.

One important feature of the theory learned by Mio is that the clauses can be applied independently of the size of the board and the number of mines. The clauses vary in complexity. Rule S-1 and Rule M-1 correspond to the trivial situations where a determined tile needs k mines and k mines are already marked, and where a determined tile needs k mines and it has k blank neighbours, respectively.

On the other hand, Rule S-3 can be seen as one of the most complex clauses because it involves three determined tiles to deduce a safe tile. Figure 8.6 left shows a board state where Rule S-3 is the only one which allows to identify a safe tile. The clause obtains the zoneOfInterest corresponding to the undetermined tile considered (Figure 8.6 center). Then by applying difference operations on the sets determined by three uncovered tiles from the zoneOfInterest (see Figure 8.6 right), the set $([40], 0)$ is obtained and thus it is deduced that tile 40 is safe.

Rules about safe tiles

```
 %% RULE S-1 %%
safe(TILEUK,BOARD):-
 zoneOfInterest(TILEUK,BOARD,ZONE),
 setHasXMines(TILEK,BOARD,ZONE,SET),
 inSet(TILEUK,SET), minesInSet(SET,0).
 %% RULE S-2 %%
safe(TILEUK,BOARD):-
  zoneOfInterest(TILEUK,BOARD,ZONE),
  setHasXMines(TILEK0,BOARD,ZONE,SET0),
  setHasXMines(TILEK1,BOARD,ZONE,SET1),
  diffSetHasXMines(SET1,SET0,SET3),
  inSet(TILEUK,SET3), minesInSet(SET3,0).
 %% RULE S-3 %%
safe(TILEUK,BOARD):-
  zoneOfInterest(TILEUK,BOARD,ZONE),
  setHasXMines(TILEK0,BOARD,ZONE,SET0),
  setHasXMines(TILEK1,BOARD,ZONE,SET1),
  setHasXMines(TILEK2,BOARD,ZONE,SET2),
  diffSetHasXMines(SET1,SET0,SET3),
  diffSetHasXMines(SET2,SET3,SET4),
  inSet(TILEUK,SET4), minesInSet(SET4,0).
 %% RULE S-4 %%
safe(TILEUK,BOARD):-
  allMinesInFringe(BOARD,SET),
  unknownNotInSet(TILEUK,BOARD,SET).
```

Rules about mines

```
 %% RULE M-1 %%
mine(TILEUK,BOARD):-
  zoneOfInterest(TILEUK,BOARD,ZONE),
  setHasXMines(TILEK,BOARD,ZONE,SET),
  inSet(TILEUK,SET), minesInSet(SET,INT),
  lengthSet(SET,INT).
```

Table 8.3: Minesweeper playing strategy learned by Mio

Rule S-4 is a clause for end-game situations. Basically it says that if all the remaining mines are in the fringe, any unknown tile outside the fringe is safe.

8.4.3 Game Playing

To evaluate the performance at game playing of each set of clauses obtained, we used each set of clauses as the playing strategy of an automatic Minesweeper player and calculated the percentage of games won by the player in 1000 random games (see Table 8.2). The playing conditions were the same as the ones presented to the human players; i.e., at the beginning the player is presented with an empty 8×8 board with $M = 10$ mines and can uncover a mine in the first move. Note that in most Minesweeper implementations, one never hits a mine in the first move.

In addition, we examined the effect of adding probabilistic reasoning. In the experiment, we instructed the player using the playing strategy shown in Table 8.3 to select a tile which minimizes the probability $P(TU)$ that an undetermined tile TU is a mine when none of the clauses can be applied. $P(TU)$ is equal to

$$max_{TD}(\frac{fm(TD)}{fn(TD)})$$

where TD is a determined neighbour of TU, $fm(TD)$ returns the number of mines needed by TD and $fn(TD)$ returns the number of blank neighbours of TD. Every time the player has to guess, it selects the tile which minimizes $P(TU)$. This player wins 570 of 1000 random games.

8.4.4 Experiments with Other ILP Systems

For the completeness of this work, we ran FOIL, Progol and TILDE on the the same training examples given to Mio when no active learning was performed. Progol and TILDE settings can be seen in Appendix B; default values were used for FOIL.

FOIL removes from the two clauses it learns **zoneOfInterest** and, by doing so, its clauses are not I/O-complete and cannot be used as playing strategy for a Minesweeper player. If FOIL would not have removed **zoneOfInterest** from the clauses, its first clause would have been identical to Rule S-1 in Table 8.3, and its second clause would have been overly general.

Progol warns about depth and resolution-bound failure, although the maximum stack depth and resolutions steps were set to 50000 and 10000, respectively;

and it learns only overly general clauses which if used as playing strategy of a Minesweeper player have an average performance below 1%.

Finally, TILDE, due to hill-climbing search myopia, returns an empty tree in four of the runs. In the fifth run, TILDE learns Rule S-4 in Table 8.3 which is applicable only in end-game situations. To alleviate TILDE's myopia, we included **zoneOfInterest** in the root of the tree and gave TILDE several lookahead-templates. With this setting, TILDE was able to learn a theory on every run; however, TILDE's theories have also an average playing performance below 1%.

These observations might imply that the optimizations (macro-operators and active inductive learning) included in Mio are indeed necessary to learn a Minesweeper playing strategy.

8.5 Related Work

Work related to learning Minesweeper playing strategies with multirelational learning can be classified into two categories, one is work done on implementing Minesweeper playing programs and the other is ILP applied to games.

8.5.1 Minesweeper Playing Programs

There are several Minesweeper programs available on the web. These programs are not learning programs but playing programs where the authors have embeded their own game playing strategy. Among these programs, John D. Ramsdell's PGMS is quite successful winning 60% of 10000 random games in a 8×8 board with 10 mines.

PGMS plays using the *Equation Strategy* based on finding approximate solutions to derived integer linear equations, and probabilities. As mentioned by Ramsdell [93], PGMS represents the information available on the board as a set of integer linear equations. Associated with an undetermined tile is a variable x that has the value 1 if the tile hides a mine, or 0 otherwise. An equation is generated for each uncovered tile with an adjacent undetermined tile. Each equation has the form $c = \sum_{i \in S} x_i$, where S is a set of undetermined tiles, and c is the number of mines hidden among S. To simplify notation, this equation is written as $c \doteq S$. Since the total number of hidden mines is known, an additional equation simply equates this number with the sum of all of the undetermined tiles.

Every time a tile t is determined safe or a mine, the board changes are propagated to all the equations containing t and a new equation for the undetermined neighbours of t is added. In addition, if $c_0 \doteq S_0$ and $c_1 \doteq S_1$ are two equations such that S_0 is a proper subset of S_1, the equation $c_1 - c_0 \doteq S_1 \setminus S_0$ is added. To determine whether a tile is safe or a mine, PGMS iteratively applies the following rules until none are applicable[7]:

- If $0 \doteq S$, all tiles in S are safe.
- If $c \doteq S$ and $c = |S|$, all tiles in S are a mine.
- Let $c_0 \doteq S_0$ and $c_1 \doteq S_1$ be two equations and t_u be an undetermined tile such that $c_0 < c_1$, and $t_u \in S_0$ and $t_u \in S_1$. If $c_1 - c_0 = |S_1 \setminus S_0|$, all the tiles in $S_1 \setminus S_0$ are a mine and all the tiles in $S_0 \setminus S_1$ are safe.

PGMS must guess when presented with a board to which none of the rules apply. For each tile t it computes the value $P(t)$ as follows. Given an equation $c \doteq S$, define its single equation probability to be $c/|S|$. $P(t)$ is equal to $max_{t \in S}(c/|S|)$. PGMS picks the tile t that minimizes $P(t)$. A random choice is made when there is more than one tile that minimizes $P(t)$.

We were surprised to notice that although Mio was only given general background knowledge about Minesweeper, the theory it learned is similar to the rules programmed in PGMS. For example, Mio's Rule S-1 and Rule S-2 correspond to the first and third rule in PGMS, respectively; and Mio's Rule M-1 is similar to PGMS second rule. To compare PGMS performance with the performance of Mio's best playing strategy, we let our best player (i.e., the player using the theory in Table 8.3 and probabilities) play 10000 random games in a 8×8 board with 10 mines. Its winning rate is also 60%.

8.5.2 Multirelational Learning for Games

Ramon et al. [91] use TILDE to learn a theory that predicts the value of candidate moves in certain Go problems. This theory is then used to reduce the number of moves considered by alpha-beta search. In [92], Ramon et al. apply TILDE for opponent modeling in Go. Opponent modeling is done with two different purposes: to characterize the play of a particular opponent and to recognize an unknown opponent.

Nakano et al. [75] presented an approach to generate an evaluation function for Shogi (Japanese chess) mating problems, which are puzzles where the player has to check continuously. The evaluation function obtained by a FOIL-like system (FOIL-I) is then used in the search to solve those problems. Finally, Morales [67] applied the system PAL to learn chess patterns to construct chess playing strategies or to analyze chess positions.

[7]In these rules $|S|$ is the cardinality of S and $S_0 \setminus S_1$ is the difference between S_0 and S_1.

The tasks addressed by the work described above are, to a certain extent, less ambitious than the task of learning a full playing strategy for a game; however, they contribute to show that ILP is a good option for learning in games.

8.6 Summary

In this chapter we described how the use of new ILP techniques such as macros, hill-climbing using macros, and active inductive learning allow Mio to learn a Minesweeper playing strategy. This learning task proved itself to be a challenging testbed for general purpose multirelational learning systems such as FOIL, Progol and TILDE.

The best theory obtained by Mio wins 53.6% of the games in a 8×8 board with 10 mines, while on average a non-expert human player wins 35% of the games. The performance of the playing program using this theory as playing strategy improves to 60% when adding the use of probabilities.

9 Conclusions and Future Work

Inductive logic programming or multirelational learning is a sub-area of machine learning concerned with inducing concept definitions using a first-order (relational) representation. Most interesting multirelational domains contain structural and/or topological information. Structural and topological data refers to the parts, arrangement and composition of complex entities or objects, and to geometrical and spatial relations among them. Dealing well with structural/topological domains is then an important issue in ILP. In this thesis we analyzed the challenges that structural/topological domains pose to current multirelational learning systems and lay the foundations to allow ILP systems to better cope with these challenges. In this chapter we summarize our contributions and give pointers to future work.

9.1 Contributions Summary

Since learning is in fact performed by searching through a hypothesis space to find the hypothesis that best meets a specific criteria, our work can be seen as improving the way ILP systems search for hypothesis in structural/topological domains. Our contributions in this respect are:

- **Thesis contribution: Macro-operators.**
  Achievement: Significant reduction of the search space explored.

  Macros are the first safe, general, effective and user-friendly formal method to reduce the search space explored by multirelational learning systems in structural/topological domains. Macros are *safe* because they discard only clauses which cannot belong to an adequate theory; they are *general* because they are suitable for every structural/topological domain; *effective* because, according to our empirical study on six application domains, macros reduce on average in a 64% the search space explored by a system; and *user-friendly* because they do not imply extra work for the user. In addition, macros reduce the search space explored by a system without further restricting the underlying hypothesis space. The algorithms provided for the macros generation are shown to be complete and correct.

- **Thesis contribution: Macro-operators.**
  Achievement: Reduction of the myopia of hill-climbing search.

  A greedy learner using macros reports, according to our empirical results, significantly lower classification error than other systems using other techniques, such as fixed-depth lookahead, template-based lookahead or determinate literals, to alleviate the myopia problem of hill-climbing search. In addition, macros, contrary to template-based lookahead, do not require user-defined templates and are automatically constructed; macros are more general than determinate literals since they are suitable for every structural/topological domain; and, contrary to beam-search and fixed-depth lookahead, macros' performance is less sensitive to domain-dependent tuning of parameters.

- **Thesis contribution: Stochastic parallel search**
  Achievement: Increasing the stability of example-driven learning.

  Stochastic selection of examples and parallel search was empirically shown to be a viable solution to increase the stability of example-driven systems against reordering of the training data. However, in our implementation, parallel search has the drawbacks of increasing the running time of the system and of being unsuitable for domains with numerical data.

- **Thesis contribution: Active inductive learning**
  Achievement: Improving the efficiency of the instance space exploration.

  Active inductive learning allows a system to gather examples of those situations not yet covered by the clauses already learned and to validate the clauses already obtained. By performing active inductive learning, a system requires less examples to learn a theory and avoids overfitting.

To measure the effectiveness of these contributions, we implemented Mio, an example-driven covering system. According to our experiments, Mio's learning results with respect to predictive accuracy can be favourably compared with that of the most commonly used systems in ILP publications.

A side contribution of this thesis is the introduction of the task of learning a Minesweeper playing strategy as a testbed for current ILP systems in structural/topological domains. This task was an arduous test for the three systems most commonly used in ILP publications. In fact, none of these systems was able to learn a playing strategy for Minesweeper. On the other hand, the use of macros and active inductive learning allows Mio to induce a Minesweeper playing strategy whose winning rate (60%) is better than the average winning rate of non-expert human players (35%) and identical to that of hand-coded playing strategies.

9.2 Future Research Directions

To establish macro-operators as a truly general approach for multirelational learning, one has to implement and evaluate them within other multirelational learning approaches. Thus, future research avenues are to adapt the use of macros to the generate-then-test approach and to upward refinement operators. For generate-then-test systems, the only needed modification is to base the macros construction on the mode declarations and eliminate the use of a bottom clause. For upward refinement operators, macros may be used to select which literals should be removed (instead of added) from a clause $\mathcal{C}$ to obtain generalizations of $\mathcal{C}$ which are legal subsequences of literals.

Recently Lavrač and Flach [58] have shown how propositionalization methods may deal with nondeterminate individual-centered domains. However, their approach is restricted to binary dependent providers denoting structural relations in individual-centered domains. Since macros relax these limitations, other future work concerning macros is to explore their use as first-order features for propositionalization approaches. In this case, after macros are constructed, a method to eliminate irrelevant macros such as the one described in [58] could be applied to reduce the number of macros (features) available.

With respect to parallel search, we observed that in domains with numerical data, parallel search is not enough to increase the stability of example-driven learning, because numerical constants appearing in the clauses are obtained from single examples. We expect that, by combining parallel search with discretization, perfect stability could be achieved in domains with numerical data.

Active inductive learning proved itself to be critical for learning a Minesweeper playing strategy; however, as future work, this framework has to be validated in other domains where minority cases are presented and there is a rich amount (thousands) of examples available.

Finally, there is still work to do to achieve the goal of finding a complete playing strategy for Minesweeper, since with Mio's playing strategy there are still board situations where the player guesses when a certain choice can still be made.

Thus, it remains to explore whether other machine learning approaches such as multiagent learning induce more complete Minesweeper playing strategies.

A Mio: User's Manual

Mio is an example-driven multirelational learning system which performs a top-down search in the subsumption lattice, which is lower bound by a bottom clause. This bottom clause is constructed by inverse entailment. Three search strategies are included in Mio: IDA*, beam-search and hill-climbing. In addition, Mio supports the use of macro-operators and is able to perform parallel search and active inductive learning. This manual explains how to use Mio but not the research results behind it; for information about macros, parallel search, active inductive learning and experiments performed with Mio the reader is referred to [79, 80, 81, 82].

A.1 Input File

Mio expects as input a single file with the user-defined parameters, mode declarations, background knowledge and training examples. Testing data may be given in an additional file.

A.1.1 Parameters

User-defined parameters, if given, should be provided one per line in the following format:

$set(*parameter*,*value*)**@**

where parameter can be any of the abbreviations listed below and value can be either an integer, a long or a float value. We describe now the parameters available in Mio.

- **General Behaviour**

  seed – sets the seed of the random number generator used to stochastically select the positive examples to guide the search. Default value: current time in milliseconds.

  verbose – controls the amount of output Mio generates. It can be an integer value between 0 and 4. Default value: 1.

- **Learning Restrictions**

  iglb – determines the variable depth of the bottom clause. The default value is automatically computed at run time so that every predicate defined in the mode declarations may be used at least once.

  mbl – specifies the maximum number of body literals allowed in any clause. Default value: 20 literals.

  mec – sets the minimum number of positive examples that a clause has to cover to be considered a solution. Default value: number of literals in the body of the clause.

  noise – indicates the maximum percentage of negative examples that a clause is allowed to cover and still be considered as a solution. Default value: 0.0%.

- **Search Settings**

  bsze – specifies the beam-width. Default value: 1 clause.

  lka – sets the amount of lookahead to perform. Default value: 1.

  nodes – determines the maximum number of clauses considered during search. Default value: 70000 clauses.

  styp – specifies the search strategy; 0 indicates IDA* and 1 hill-climbing search. Default value: 0.

- **Parallel Search**

  msp – specifies the minimum number of remaining positive examples taken for parallelization. Default value: 1 example.

  mxp – sets the maximum number of remaining positive examples taken for parallelization. Default value: 500 examples.

  pp – determines the percentage of remaining positive examples taken for parallelization. Default value: 0.0%.

- **Active Inductive Learning**

  active – sets active inductive learning; the value given in the **set** command is ignored.

  almxi – indicates the maximum number of active inductive learning iterations to be performed. Default value: 3 iterations.

  alt – determines with how many remaining positive examples asks the inductive learner for more examples to the server. Default value: 1 remaining positive example.

nne – indicates the number of negative examples the active inductive server should supply to the learner. Default value: 20 negative examples.

npe – indicates the number of positive examples the active inductive server should supply to the learner. Default value: 20 positive examples.

lap – sets the minimum acceptable Laplace value of a clause. Any clause with a Laplace estimate below this threshold is discarded. The Laplace estimate of a clause $\mathcal{C}$ is calculated by $(p+1)/(p+n+2)$ where p and n are the number of positive and negative examples covered by $\mathcal{C}$. Default value: 0.50.

pur – sets the minimum acceptable purity value of a clause. Any clause with a purity estimate below this threshold is discarded. The purity of a clause $\mathcal{C}$ is calculated by $p/(p+n)$ where p and n are the number of positive and negative examples covered by $\mathcal{C}$. Default value: 0.99.

seedge – sets the seed of the random number generator used to generate the examples. Default value: current time in milliseconds.

tsz – indicates the minimum number of examples used by the active server to test the learner's solutions. After testing the clauses once, the server checks whether a request from the learner has been received. If no request has been received, it performs a new testing iteration. Up to 15 testing iterations may be executed. Default value: 40 examples.

A.1.2 Mode Declarations

The mode declarations define the literals that may appear in the head (target concept) and in the body of any learned clause. In addition, they provide Mio with information about the type and mode of the literal's arguments. Mode declarations have to be provided, one per line, in the following formats:

\$modeh(r**, target_concept(**m_1**type**$_1$**,...,**m_n**type**$_n$**))@**
\$modeb(r**, p(**m_1**type**$_1$**,...,**m_n**type**$_n$**))@**

m_i is the mode of a variable argument and can be one of the following four modes:

- $+X$ indicates that X is an input argument; X is bound to a variable appearing in a literal already in the clause.
- $-X$ means that X is an output argument; i.e., X is a new variable.
- $*X$ means that X is an output argument of a literal which is a dependent provider.
- $\#X$ indicates that X has to be replaced by a constant value in the literal.

type$_i$ indicates the type of the ith argument. The background knowledge should contain a predicate definition of **type**$_i$, or **type**$_i$ should be a SICStus built-in predicate, so that, the query **:-type**: i**(X)** can be executed.

r is an integer value which indicates the maximal number of different bindings for the output variables of a literal, which may be computed given a specific binding for its input variables.

A.1.3 Background Knowledge

The background knowledge is given as a definite program. Since Mio uses as Prolog engine SICStus Prolog, any logic program executable by SICStus can be given as background knowledge. This also means that all SICStus built-in predicates and libraries can be used in the background knowledge.

A.1.4 Examples

Examples are provided as ground facts in the input file. Each example should end with a new line character. Negative examples are identified with the symbol '>'. For instance, the following five examples are syntactically correct examples for the chess moves application domain.

```
move(bishop,pos(4,3),pos(6,5)).
move(bishop,pos(4,8),pos(5,7)).
move(bishop,pos(5,4),pos(4,5)).
>move(knight,pos(2,8),pos(2,6)).
>move(knight,pos(2,8),pos(3,7)).
```

Testing examples can be supplied to Mio in a separate file following the format described above.

A.2 Running Mio

Mio is implemented in Java™ under Solaris and is distributed as a JAR file. To run it, use the command:

java -jar *java-options* **Mio.jar** *input-file* [*test-file*]

Some useful Java-options are -Xms and -Xmx which set the initial and maximum Java heap size, respectively. The default maximum Java heap size is 16MB which depending on the number of examples and background knowledge may be insufficient.

Let us illustrate how to run Mio. Assume we have our input-file *in.pl* which contains the parameters, mode declarations, background knowledge, and training examples, and which is placed in the directory testfiles. This directory is under the directory where the Mio.jar file is. In addition, we have our testing examples in the file *test.pl* located at /common/ILP/testData, and want to allocate for Mio up to 64MB of memory. Then we run Mio with the following command:

java -jar -Xmx64mb Mio.jar testfiles/in.pl /user/ILP/test.pl

A.3 Mio's Output

Mio's output goes to standard output. With a verbose value of 0, Mio prints the number of examples read, the value of the user-defined parameters, and the learning and testing results. Below follows an example of Mio's output at the 0 verbosity level.

```
Input File: testfiles/fam.pl Writing in File: tmp/tmp1070540670776.pl
2 Positive examples
3 Negative examples
-- listing properties --
iglb=4.0
msp=1.0
verbose=0.0
styp=0.0
mec=1.0
--- Run time information ---
Pseudorandom Generator Seed= 1070540671162
USING IDA* SEARCH

 *** SOLUTIONS (1):
auntOf(PERSON0,PERSON1):- parentOf(PERSON0,PERSON3),
sisterOf(PERSON3,PERSON1).
Total learning time in ms: 2511
Total time elapsed in ms: 2532
```

With verbose set to one, the output includes bottom clauses constructed and solutions found at each covering iteration. The search space is also printed with a verbose value of 2. At the third verbosity level, Mio additionally outputs the macros generated. In the highest verbosity level (i.e., 4), one sees as well the bottom clause construction iterations and the variable bindings.

A.4 FAQ

1. **Can Mio learn multiple predicates?**

   Yes, you just need to give Mio a **modeh** declaration for each target concept and the corresponding negative and positive examples.

2. **Can Mio learn recursive clauses?**

   Yes; however, Mio recursive capabilities have been only tested in simple domains such as factorial, quick sort and member. In addition, when learning recursive clauses, Mio ignores any testing data provided. As an example of an input file to learn recursive clauses, here are the settings, mode declarations and background knowledge used for quick sort.

```
$set(iglb,5)@
$set(mbl,5)@
$set(styp,1)@

$modeh(1, qsort(+clist,-clist))@ % target concept
$modeb(1, components(+clist,*const,*clist))@
$modeb(1, partition(+const,+clist,*clist,*clist))@
$modeb(1, qsort(+clist,*clist))@
$modeb(1, insert(+clist,+const,+clist,*clist))@

%const type definition
const(0).  const(1).  const(2).  const(3).

%%% clist is extensionally defined as all lists up
%%% to 4 digits made with 0,1,2,and 3; e.g.,
clist([0,1,2,3]).
clist([1,0,2,3]).
 %% etc...
clist([3]).
clist([]).

:- [library(lists)]. % A SICStus library
components([H|Tail], H, Tail):-!.

insert(L1, Cte, L2, L3):-
   Ltemp = [Cte|L2], append(L1,Ltemp,L3).

partition(_,[],[],[]).
partition(X,[Y|Tail],[Y|Small],Big):-
   X > Y, !, partition(X,Tail,Small,Big).
```

```
partition(X,[Y|Tail], Small, [Y|Big]):-
   partition(X,Tail,Small,Big).
```

3. **How do I run Mio with active inductive learning?**

   Active inductive learning in Mio is implemented as a Java RMI (Remote Method Invocation) Application, which means that there are two independent Java applications: the active server and Mio. More information about Java RMI Applications can be found at http://java.sun.com/docs/books/tutorial/rmi/overview.html

   To run Mio with active inductive learning, you have to do the following:

   a) Create a class ExampleGenerator which implements the interface Generator provided with Mio distribution package.
   b) Compile the class ActiveLearningServer
   c) Write a Java security policy file and install it in Mio's active server directory and Mio's directory (a sample policy file is provided).
   d) Run the Java RMI remote object registry (rmiregistry) in the server's directory.
   e) Start the server with the command:
      java -jar -Djava.security.policy=java.policy ActiveLearningServer
      If everything is fine, the message "ActiveLearningServer ready" is shown.
   f) In Mio's input file, set the parameters accordingly to perform active inductive learning.
   g) Run Mio with the command:
      java -jar -Djava.security.policy=java.policy Mio *input-file*
      If necessary, extra memory can be allocated for Mio. In that case, use the command:
      java -jar -Djava.security.policy=java.policy -Xmx*64***mb Mio** *input-file*

B Settings

B.1 Mio Settings per Dataset

B.1.1 Chess Moves

```
$set(mbl,2)@
$set(iglb,2)@
$set(verbose,1)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

%Parameters for parallel search
%Set percentage of positive examples taken for parallelization
%$set(pp,5)@
%Set minimum number of examples taken for parallelization
%$ set(msp,4)@

% Target concept
$ modeh(1,move(#piece,pos(+column,+row),pos(+column,+row)))@

%%% Mode declarations for macro-based approach
%%% Comment out for literal-based approach
$ modeb(1,diff(+row,+row,*integer))@
$ modeb(1,diff(+column,+column,*integer))@
%%% Uncomment for literal-based approach
%$ modeb(1,diff(+row,+row,-integer))@
%$ modeb(1,diff(+column,+column,-integer))@

%%% Mode declarations common to both approaches
```

```
$ modeb(1,diff(+row,+row,+integer))@
$ modeb(1,diff(+column,+column,+integer))@
$ modeb(1,diff(+row,+row,#integer))@
$ modeb(1,diff(+column,+column,#integer))@
```

B.1.2 Eastward Trains

```
$set(iglb,3)@
$set(mbl,4)@
$set(mec,2)@
$set(verbose,1)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

%Parameters for parallel search
%Set percentage of positive examples taken for parallelization
%set(pp,40)@
%Set minimum number of examples taken for parallelization
%$set(msp, 4)@

% Target concept
$ modeh(1,east(+train))@

%%% Mode declarations for macro-based approach
%%% Comment out for literal-based approach
$ modeb(10,has_car(+train,*car))@
$ modeb(10,infront(+train,*car,*car))@
$ modeb(10,infront(+train,+car,*car))@
$ modeb(10,infront(+train,*car,+car))@

%%% Uncomment for literal-based approach
%$ modeb(10,has_car(+train,-car))@
%$ modeb(10,infront(+train,-car,-car))@
%$ modeb(10,infront(+train,+car,-car))@
%$ modeb(10,infront(+train,-car,+car))@

%%% Mode declarations common to both approaches
$ modeb(1, shape(+car, #shape))@
$ modeb(1,long(+car))@
$ modeb(1,closed(+car))@
$ modeb(1,short(+car))@
$ modeb(1,open(+car))@
$ modeb(1,double(+car))@
$ modeb(1,jagged(+car))@
$ modeb(1,flat(+car))@
$ modeb(1,peaked(+car))@
$ modeb(1,arc(+car))@
$ modeb(1,sload(+car,#shape))@
$ modeb(1,nload(+car,#integer))@
$ modeb(1,wheels(+car,#integer))@
```

B.1.3 Student Loan

```
$set(iglb, 3)@
$set(mbl, 4)@
$set(verbose,1)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

%Parameters for parallel search
%Set percentage of positive examples taken for parallelization
%$set(pp, 50)@
%Set minimum number of examples taken for parallelization
%$set(msp, 4)@

%Target concept
$modeh(1, no_payment_due(+student))@

%%% Mode declarations for macro-based approach
%%% Comment out for literal-based approach
$modeb(1, longest_absence_from_school(+student, *integer))@
$modeb(2, enrolled(+student,*school,*integer))@

%%% Mode declarations for literal-based approach
%$modeb(1, longest_absence_from_school(+student, -integer))@
%$modeb(2, enrolled(+student,-school,-integer))@

%%% Mode declarations common to both approaches
$modeb(1, male(+student))@
$modeb(2, enlist(+student,-org))@
$modeb(1, armed_forces(+org))@
$modeb(1, peace_corps(+org))@
$modeb(1, unemployed(+student))@
$modeb(1, filed_for_bankrupcy(+student))@
```

```
$modeb(1, disabled(+student))@
$modeb(15, gte(+integer, #integer))@
$modeb(15, lte(+integer, #integer))@
```

B.1.4 Mutagenesis

```
$ set(iglb,3)@
$ set(mbl,3)@
$ set(noise,7)@
$ set(mec,3)@
$ set(verbose,1)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

% Target concept
$ modeh(1,active(+drug))@

%%% Comment out for literal-based approach
$ modeb(1,lumo(+drug,*energy))@
$ modeb(1,logp(+drug,*hydrophob))@
$ modeb(50,bond(+drug,*atomid,*atomid,#integer))@
%%% Uncomment for literal-based approach
%$ modeb(1,lumo(+drug,-energy))@
%$ modeb(1,logp(+drug,-hydrophob))@
%$ modeb(50,bond(+drug,-atomid,-atomid,#integer))@

$ modeb(50,bond(+drug,-atomid,+atomid,#integer))@
$ modeb(50,bond(+drug,+atomid,-atomid,#integer))@
$ modeb(50,atm(+drug,-atomid,#element,#integer,-charge))@
$ modeb(10,atm(+drug,+atomid,#element,#integer,-charge))@

$ modeb(1,gteq(+charge,#float))@
$ modeb(1,gteq(+energy,#float))@
$ modeb(1,gteq(+hydrophob,#float))@
$ modeb(1,lteq(+charge,#float))@
$ modeb(1,lteq(+energy,#float))@
$ modeb(1,lteq(+hydrophob,#float))@
$ modeb(1,equal(+charge,#charge))@
$ modeb(1,equal(+energy,#energy))@
$ modeb(1,equal(+hydrophob,#hydrophob))@

%%% Comment out for literal-based approach
```

```
$ modeb(50,benzene(+drug,*ring))@
$ modeb(50,ring_size_6(+drug,*ring))@
$ modeb(50,nitro(+drug,*ring))@
%%% Uncomment for literal-based approach
%$ modeb(50,benzene(+drug,-ring))@
%$ modeb(50,ring_size_6(+drug,-ring))@
%$ modeb(50,nitro(+drug,-ring))@

$ modeb(50,carbon_5_aromatic_ring(+drug,-ring))@
$ modeb(50,carbon_6_ring(+drug,-ring))@
$ modeb(50,hetero_aromatic_6_ring(+drug,-ring))@
$ modeb(50,hetero_aromatic_5_ring(+drug,-ring))@
$ modeb(50,ring_size_5(+drug,-ring))@
$ modeb(50,methyl(+drug,-ring))@
$ modeb(50,anthracene(+drug,-ringlist))@
$ modeb(50,phenanthrene(+drug,-ringlist))@
$ modeb(50,ball3(+drug,-ringlist))@
$ modeb(1,member(+ring,+ringlist))@
$ modeb(1,connected(+ring,+ring))@
```

B.1.5 Mesh Design

```
$set(mbl,5)@
$set(noise,1)@
$set(mec,2)@
$set(iglb,3)@
$set(verbose,1)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

%Parameters for parallel search
%Set percentage of positive examples taken for parallelization
%$set(pp, 70)@
%Set minimum number of examples taken for parallelization
%$set(msp, 10)@

% Target concept
$modeh(1, mesh(+edge,#integer))@

$modeb(1,long(+edge))@
$modeb(1,usual(+edge))@
$modeb(1,short(+edge))@
```

```
$modeb(1,circuit(+edge))@
$modeb(1,half_circuit(+edge))@
$modeb(1,quarter_circuit(+edge))@
$modeb(1,short_for_hole(+edge))@
$modeb(1,long_for_hole(+edge))@
$modeb(1,circuit_hole(+edge))@
$modeb(1,half_circuit_hole(+edge))@
$modeb(1,not_important(+edge))@
$modeb(1,free(+edge))@
$modeb(1,one_side_fixed(+edge))@
$modeb(1,two_side_fixed(+edge))@
$modeb(1,fixed(+edge))@
$modeb(1,not_loaded(+edge))@
$modeb(1,one_side_loaded(+edge))@
$modeb(1,two_side_loaded(+edge))@
$modeb(1,cont_loaded(+edge))@
$modeb(6,opposite(+edge,-edge))@

%%% Macro-based approach
%%% Comment out for literal-based approach
$modeb(6,neighbour(+edge,*edge))@

%%% Uncomment for literal-based approach
%$modeb(6,neighbour(+edge,-edge))@
```

B.1.6 Traffic Problem Detection

```
$set(verbose,1)@
$set(iglb,3)@
$set(mbl,4)@
$set(mec,2)@
%$set(styp,1)@  % Uncomment to set hill-climbing search
%$set(lka,2)@   % Uncomment to set lookahead with hill-climbing (s=2)
%$set(bsze,10)@ % Uncomment to set beam-search (b=10)

% Target concepts
$modeh(1,accident(+section,+time))@
$modeh(1,congestion(+section,+time))@
$modeh(1,noncs(+section,+time))@

%%% Mode declarations common to both approaches
$modeb(1,velocidadd(+time,+section,#velocity))@
$modeb(1,ocupaciond(+time,+section,#velocity))@
```

```
$modeb(1,saturaciond(+time,+section,#velocity))@
$modeb(1,tipo(+section,#stype))@

%%% Mode declarations for macro-based approach
%%% Comment out for literal-based approach
$modeb(100,secciones_posteriores(+section,*section))@
$modeb(100,secciones_posteriores(*section,+section))@

%%% Uncomment for literal-based approach
%$modeb(100,secciones_posteriores(+section,-section))@
%$modeb(100,secciones_posteriores(-section,+section))@
```

B.1.7 Minesweeper

```
%% Active inductive learning parameters
%% Comment out to unset active inductive learning
$set(active,1)@
$set(nne,10)@
$set(npe,10)@
$set(alt,1)@
$set(tsz,50)@
$set(pur,0.99)@
$set(almxi, 3)@
$set(seedge, 1034604377230)@  % Example generator random seed
$set(seed, 1034343047909)@    % Examples selection random seed

%% General parameters
$set(iglb,5)@
$set(mec,1)@
$set(verbose,1)@
$set(mbl,9)@
$set(set,1)@
$set(nodes,4000)@
%$set(styp,1)@  % Uncomment to perform Hill-climbing search
%$set(lka,2)@   % Uncomment to perform lookahead (s=2)

$modeh(1, safe(+tileUK,+board))@ %Target Concept

%%% Mode declarations for macro-based approach
%%% Comment out for literal-based approach
$modeb(1, zoneOfInterest(+tileUK, +board, *zone))@
$modeb(1, totalMinesLeft(+board,*integer))@
$modeb(1, allMinesInFringe(+board, *set))@
```

```
$modeb(24, setHasXMines(*tileK, +board, +zone, *set))@
$modeb(1, diffSetHasXMines(+set, +set, *set))@
$modeb(1, unknownNotInSet(+tileUK, +board, +set))@
$modeb(1, inSet(+tileUK, +set))@
$modeb(1, lengthSet(+set,+integer))@
$modeb(1, minesInSet(+set,*integer))@
$modeb(1, minesInSet(+set,#integer))@

%%% Uncomment for  literal-based approach
%$modeb(1, zoneOfInterest(+tileUK, +board, -zone))@
%$modeb(1, totalMinesLeft(+board,-integer))@
%$modeb(1, allMinesInFringe(+board, -set))@
%$modeb(24, setHasXMines(-tileK, +board, +zone, -set))@
%$modeb(1, diffSetHasXMines(+set, +set, -set))@
%$modeb(1, unknownNotInSet(+tileUK, +board, +set))@
%$modeb(1, inSet(+tileUK, +set))@
%$modeb(1, lengthSet(+set,+integer))@
%$modeb(1, minesInSet(+set,-integer))@
%$modeb(1, minesInSet(+set,#integer))@
```

B.2 TILDE Settings per Dataset

B.2.1 Chess Moves

```
predict(move(+mid,-class)).

typed_language(yes).
type(position(mid, c,r)).
type(position(mid, c,r)).
type(diffCols(mid,number)).
type(diffRows(mid,number)).
type(piece(mid,p)).

rmode(1: piece(+M, #[king,queen,rook,bishop,knight])).
rmode(1: diffCols(+M,-N)).
rmode(1: diffRows(+M,-N)).
rmode(1: diffRows(+M,#[0,1,2,3,4,5,6,7])).
rmode(1: diffCols(+M,#[0,1,2,3,4,5,6,7])).

%%% Uncomment to activate lookahead
%lookahead(diffRows(M,N), diffCols(M,N)).
%lookahead(diffCols(M,#[0,1,2,3,4,5,6,7]),
%          diffRows(M,#[0,1,2,3,4,5,6,7])).
%lookahead(diffRows(M,N), diffCols(M,#[0,1,2,3,4,5,6,7])).
%lookahead(diffCols(M,N), diffRows(M,#[0,1,2,3,4,5,6,7])).
```

B.2.2 Eastward Trains

```
predict(train(+tid,-class)).
classes([east,west]).

typed_language(yes).
type(cars(tid, lcars)).
type(has_car(tid,car)).
type(infront(tid,car,car)).
type(shape(car,shape)).
type(has_roof(car,roof)).
type(long(car)).
type(double(car)).
type(short(car)).
type(closed(car)).
type(sload(car,shape)).
type(nload(car,number)).
type(wheels(car,number)).

rmode(10: has_car(+TID, -Car)).
rmode(10: infront(+TID,-Car1,-Car2)).
rmode(1: #(5*5*S: shape(C,S), shape(+C,S))).
rmode(1: long(+Car)).
rmode(1: double(+Car)).
rmode(1: short(+Car)).
rmode(1: closed(+Car)).

% Following four mode declarations have to
% be commented out when performing lookahead
rmode(1: #(5*1*R: has_roof(C,R), has_roof(C,R))).
rmode(1: #(5*5*S: sload(C,S), sload(+C,S))).
rmode(1: #(5*5*N: nload(C,N), nload(+C,N))).
rmode(1: #(5*5*N: wheels(C,N), wheels(+C,N))).
% Uncomment to perform lookahead
%rmode(1: (has_car(+TID, -Car), long(+Car))).
%rmode(1: (has_car(+TID, -Car), double(+Car))).
%rmode(1: (has_car(+TID, -Car), short(+Car))).
%rmode(1: (has_car(+TID, -Car), closed(+Car))).
```

B.2.3 Student Loan

```
predict(no_payment_due(+sid,-class)).
classes([pos,neg]).

typed_language(yes).
type(male(sid)).
type(longest_absence_from_school(sid,number)).
type(enrolled(sid,school,number)).
type(enlist(sid,org)).
type(armed_forces(org)).
type(peace_corps(org)).
type(unemployed(sid)).
type(filed_for_bankrupcy(sid)).
type(disabled(sid)).
type(gte(number,number)).
type(lte(number,number)).

rmode(1: male(+SID)).
rmode(1: longest_absence_from_school(+SID, -N)).
rmode(1: enrolled(+SID,-S,-N)).
rmode(1: enlist(+SID, -O)).
rmode(1: armed_forces(+O)).
rmode(1: peace_corps(+O)).
rmode(1: unemployed(+SID)).
rmode(1: filed_for_bankrupcy(+SID)).
rmode(1: disabled(+SID)).
rmode(1: #(10*1*N: longest_absence_from_school(SID, N), gte(+N1,N))).
rmode(1: #(10*1*N: longest_absence_from_school(SID, N), lte(+N1,N))).
rmode(1: #(10*1*N: enrolled(SID, S, N), gte(+N1,N))).
rmode(1: #(10*1*N: enrolled(SID, S, N), lte(+N1,N))).

% Uncomment to activate lookahaed
%lookahead(enrolled(SID, S, N),  gte(+N1,N)).
%lookahead(enrolled(SID, S, N),  lte(+N1,N)).
%lookahead(longest_absence_from_school(SID, N), gte(+N1,N)).
%lookahead(longest_absence_from_school(SID, N), lte(+N1,N)).
```

B.2.4 Mutagenesis

```
%%% Settings obtained from~\cite{blockeel}
load(models).
```

```
discretization(bounds(3)).
to_be_discretized(lumo(X), [X]).
to_be_discretized(logp(X), [X]).
to_be_discretized(atom(_,_,_,X), 100, [X]).

typed_language(yes).
type(atom(id, element, type, charge)).
type(bond(id, id, bondtype)).
type(lumo(lumo)).
type(logp(logp)).
type(X =< X).
type(X=X).

type(nitro(structure)).
type(carbon_6_ring(structure)).
type(benzene(structure)).
type(ring_size_6(structure)).
type(ring_size_5(structure)).
type(methyl(structure)).
type(phenanthrene(structure)).
type(anthracene(structure)).
type(ball3(structure)).
type(hetero_aromatic_5_ring(structure)).
type(hetero_aromatic_6_ring(structure)).
type(carbon_5_aromatic_ring(structure)).
type(occurs_in(id, structure)).

type(discretized(_, _, _)).
type(member2(_, _)).

max_lookahead(5).
lookahead(nitro(S), occurs_in(A, S)).
lookahead(carbon_6_ring(S), occurs_in(A, S)).
lookahead(benzene(S), occurs_in(A, S)).
lookahead(ring_size_6(S), occurs_in(A, S)).
lookahead(ring_size_5(S), occurs_in(A, S)).
lookahead(methyl(S), occurs_in(A, S)).
lookahead(phenanthrene(S), occurs_in(A, S)).
lookahead(anthracene(S), occurs_in(A, S)).
lookahead(ball3(S), occurs_in(A, S)).
lookahead(hetero_aromatic_5_ring(S), occurs_in(A, S)).
lookahead(hetero_aromatic_6_ring(S), occurs_in(A, S)).
lookahead(carbon_5_aromatic_ring(S), occurs_in(A, S)).
lookahead(occurs_in(A, S), #(230*37*T: atom(A, _, T, _),
          atom(A, E, T, Ch))).
lookahead(occurs_in(A, S), #(230*9*E: atom(A, E, _, _),
          atom(A, E, T, Ch))).

rmode(20: #(1*10*C: (discretized(lumo(X), [X], L), member2(C,L)),
          +Lumo=<C)).
rmode(20: #(1*10*C: (discretized(logp(X), [X], L), member2(C,L)),
          +LP =< C)).
rmode(20: #(230*37*T: atom(_, _, T, _), atom(A, E, T, Ch))).
rmode(20: #(230*9*E: atom(_, E, _, _), atom(A, E, T, Ch))).
rmode(20: #(1*100*C:(discretized(atom(_,_,_,X), [X], L),
          member2(C, L)), (atom(-A, E, T, Ch), Ch =< C))).
rmode(20: bond(+A1, -A2, BT)).
rmode(3: nitro(-S)).
rmode(3: carbon_6_ring(-S)).
rmode(3: benzene(-S)).
rmode(3: ring_size_6(-S)).
rmode(3: ring_size_5(-S)).
rmode(3: methyl(-S)).
rmode(3: phenanthrene(-S)).
rmode(3: anthracene(-S)).
rmode(3: ball3(-S)).
rmode(3: hetero_aromatic_5_ring(-S)).
rmode(3: hetero_aromatic_6_ring(-S)).
rmode(3: carbon_5_aromatic_ring(-S)).

root((lumo(Lumo), logp(Logp))).
```

B.2.5 Mesh Design

```
%%% Settings obtained from~\cite{blockeel}
tilde_mode(regression).
load(models).
euclid(mesh(E,X), X).
root(mesh(E,X)).

rmode(5: long(+E)).
rmode(5: usual(+E)).
rmode(5: short(+E)).
rmode(5: circuit(+E)).
rmode(5: quarter_circuit(+E)).
rmode(5: short_for_hole(+E)).
rmode(5: long_for_hole(+E)).
rmode(5: circuit_hole(+E)).
rmode(5: half_circuit_hole(+E)).
rmode(5: not_important(+E)).
rmode(5: free(+E)).
rmode(5: one_side_fixed(+E)).
rmode(5: two_side_fixed(+E)).
rmode(5: fixed(+E)).
rmode(5: not_loaded(+E)).
rmode(5: one_side_loaded(+E)).
rmode(5: two_side_loaded(+E)).
rmode(5: cont_loaded(+E)).
rmode(5: neighbour(+E,-E2)).
rmode(5: opposite(+E,-E2)).

%% Comment out to deactivate lookahead
lookahead(neighbour(E1, E2), long(E2)).
lookahead(neighbour(E1, E2), usual(E2)).
lookahead(neighbour(E1, E2), short(E2)).
lookahead(neighbour(E1, E2), circuit(E2)).
lookahead(neighbour(E1, E2), quarter_circuit(E2)).
lookahead(neighbour(E1, E2), short_for_hole(E2)).
lookahead(neighbour(E1, E2), long_for_hole(E2)).
lookahead(neighbour(E1, E2), circuit_hole(E2)).
lookahead(neighbour(E1, E2), half_circuit_hole(E2)).
lookahead(neighbour(E1, E2), not_important(E2)).
lookahead(neighbour(E1, E2), free(E2)).
lookahead(neighbour(E1, E2), one_side_fixed(E2)).
lookahead(neighbour(E1, E2), two_side_fixed(E2)).
lookahead(neighbour(E1, E2), fixed(E2)).
lookahead(neighbour(E1, E2), not_loaded(E2)).
lookahead(neighbour(E1, E2), one_side_loaded(E2)).
lookahead(neighbour(E1, E2), two_side_loaded(E2)).
lookahead(neighbour(E1, E2), cont_loaded(E2)).
lookahead(opposite(E1, E2), long(E2)).
lookahead(opposite(E1, E2), usual(E2)).
lookahead(opposite(E1, E2), short(E2)).
lookahead(opposite(E1, E2), circuit(E2)).
lookahead(opposite(E1, E2), quarter_circuit(E2)).
lookahead(opposite(E1, E2), short_for_hole(E2)).
lookahead(opposite(E1, E2), long_for_hole(E2)).
lookahead(opposite(E1, E2), circuit_hole(E2)).
lookahead(opposite(E1, E2), half_circuit_hole(E2)).
lookahead(opposite(E1, E2), not_important(E2)).
lookahead(opposite(E1, E2), free(E2)).
lookahead(opposite(E1, E2), one_side_fixed(E2)).
lookahead(opposite(E1, E2), two_side_fixed(E2)).
lookahead(opposite(E1, E2), fixed(E2)).
lookahead(opposite(E1, E2), not_loaded(E2)).
lookahead(opposite(E1, E2), one_side_loaded(E2)).
lookahead(opposite(E1, E2), two_side_loaded(E2)).
lookahead(opposite(E1, E2), cont_loaded(E2)).
```

B.2.6 Traffic Problem Detection

```
minimal_cases(3).
talking(4).
load(models).
classes([accident,congestion,noncs]).
root((section(A), timemoment(B))).

typed_language(yes).
type(section(sec)).
type(timemoment(time)).
type(tipo(sec,type)).
type(velocidadd(time,sec,range)).
type(ocupaciond(time,sec,range)).
type(saturaciond(time,sec,range)).
type(secciones_posteriores(sec,sec)).
type(velocidad(time,sec,number)).
type(saturacion(time,sec,number)).
type(ocupacion(time,sec,number)).

rmode(velocidadd(+T,+S,#[baja,media,alta])).
rmode(ocupaciond(+T,+S,#[baja,media,alta])).
rmode(saturaciond(+T,+S,#[baja,media,alta])).
rmode(tipo(+S,#[carretera,rampa_abandono,rampa_incorporacion])).
rmode(secciones_posteriores(+S,-S1)).
rmode(secciones_posteriores(-S1,+S)).

% Uncomment to activate lookahaed
%max_lookahead(2).
%lookahead(secciones_posteriores(S,S1),
%           tipo(S1,#[carretera,rampa_abandono,rampa_incorporacion])).
%lookahead(secciones_posteriores(S,S1),
%           velocidadd(+T,S1,#[baja,media,alta])).
%lookahead(secciones_posteriores(S,S1),
%           saturaciond(+T,S1,#[baja,media,alta])).
```

```
%lookahead(secciones_posteriores(S,S1),
%           ocupaciond(+T,S1,#[baja,media,alta])).
%lookahead(secciones_posteriores(S1,S),
%           tipo(S1,#[carretera,rampa_abandono,rampa_incorporacion])).
%lookahead(secciones_posteriores(S1,S),
%           velocidadd(+T,S1,#[baja,media,alta])).
%lookahead(secciones_posteriores(S1,S),
%           saturaciond(+T,S1,#[baja,media,alta])).
%lookahead(secciones_posteriores(S1,S),
%           ocupaciond(+T,S1,#[baja,media,alta])).
```

B.2.7 Minesweeper

```
minimal_cases(2).
talking(4).
use_packs(0).
output_options([c45,prolog,elaborate]).
classes([safe,mine]).

load(models).
%With the following root, TILDE generates an empty tree
root((tile(T),board(B))).
%With the next one, Tilde learns a theory
%root((tile(T),board(B),zoneOfInterest(T,B,Z))).

typed_language(yes).
type(tile(tileUK)).
type(board(board)).
type(zoneOfInterest(tileUK, board, zone)).
type(totalMinesLeft(board,integer)).
type(allMinesInFringe(board, set)).
type(setHasXMines(tileK, board, zone, set)).
type(diffSetHasXMines(set, set, set)).
type(unknownNotInSet(tileUK, board, set)).
type(inSet(tileUK, set)).
type(lengthSet(set,integer)).
type(minesInSet(set,integer)).

%Comment next line out when second root is used
rmode(1: zoneOfInterest(+TUK, +B, -Z)).
rmode(1: totalMinesLeft(+B, -X)).
rmode(1: allMinesInFringe(+B, -S)).
rmode(8: setHasXMines(-TK, +B, +Z, -S)).
rmode(8: diffSetHasXMines(+S, +S1, -S2)).
rmode(1: unknownNotInSet(+TUK, +B, +S)).
rmode(1: inSet(+TUK, +S)).
rmode(1: lengthSet(+S,+X)).
rmode(1: minesInSet(+S,-X)).
rmode(1: minesInSet(+S,#[0,1,2,3,4,5,6,7,8])).

%Comment next line out when second root is used
lookahead(zoneOfInterest(TUK, B, Z), setHasXMines(TK, B, Z, S)).
lookahead(setHasXMines(TK, B, Z, S), inSet(TUK, S)).
lookahead(setHasXMines(TK, B, Z, S), minesInSet(S, X)).
lookahead(setHasXMines(TK, B, Z, S),
          minesInSet(S, #[0,1,2,3,4,5,6,7,8])).
lookahead(setHasXMines(TK, B, Z, S), diffSetHasXMines(S, S1, S2)).
lookahead(diffSetHasXMines(S, S1, S2),  inSet(TUK, S2)).
lookahead(diffSetHasXMines(S, S1, S2),  minesInSet(S2, X)).
```

B.3 Progol Settings per Dataset

B.3.1 Chess Moves

```
:- set(c,2)?
:- set(i,1)?
:- set(verbose, 2)?

:- modeh(1,move(#piece,pos(+file,+rank),pos(+file,+rank)))?
:- modeb(1,rdiff(+rank,+rank,-nat))?
:- modeb(1,fdiff(+file,+file,-nat))?
:- modeb(1,rdiff(+rank,+rank,#nat))?
:- modeb(1,fdiff(+file,+file,#nat))?
:- commutative(rdiff/3)?
:- commutative(fdiff/3)?
```

B.3.2 Eastward Trains

```
:- set(nodes,3750)?
:- set(i,3)?
:- set(c,4)?
:- set(verbose,2)?
:- set(r,1000000)?
:- set(h,1000000)?
:- modeh(1,east(+train))?
:- modeb(100,has_car(+train,-car))?
:- modeb(100,infront(+train,-car,-car))?
:- modeb(100,infront(+train,+car,-car))?
:- modeb(100,infront(+train,-car,+car))?
:- modeb(1,shape(+car,#shape))?
:- modeb(1,long(+car))?
:- modeb(1,closed(+car))?
:- modeb(1,short(+car))?
:- modeb(1,open(+car))?
:- modeb(1,double(+car))?
:- modeb(1,jagged(+car))?
:- modeb(1,flat(+car))?
:- modeb(1,peaked(+car))?
:- modeb(1,arc(+car))?
:- modeb(1,sload(+car,#shape))?
:- modeb(1,nload(+car,#int))?
:- modeb(1,wheels(+car,#int))?
```

B.3.3 Student Loan

```
:-set(nodes, 1280)?
:-set(verbose,2)?

:-modeh(1, no_payment_due(+student))?
:-modeb(1, male(+student))?
:-modeb(1, longest_absence_from_school(+student, -nat))?
:-modeb(2, enrolled(+student,-school,-nat))?
:-modeb(2, enlist(+student,-org))?
:-modeb(1, armed_forces(+org))?
:-modeb(1, peace_corps(+org))?
:-modeb(1, unemployed(+student))?
:-modeb(1, filed_for_bankrupcy(+student))?
:-modeb(1, disabled(+student))?
:-modeb(15, gte(+nat, #nat))?
:-modeb(15, lte(+nat, #nat))?
```

B.3.4 Mutagenesis

```
:- set(i,2)?
:- set(nodes,20000)?
:- set(noise,5)?
:- set(c,3)?
:- set(verbose,1)?
:- set(h,10000)?
:- set(r,100000)?
:- noreductive?
:- unset(splitting)?

:- modeh(1,active(+drug))?

:- modeb(1,lumo(+drug,-energy))?
:- modeb(1,logp(+drug,-hydrophob))?
:- modeb(*,bond(+drug,-atomid,-atomid,#int))?
:- modeb(*,bond(+drug,+atomid,-atomid,#int))?
:- modeb(*,atm(+drug,-atomid,#element,#int,-charge))?
:- modeb(1,gteq(+charge,#float))?
:- modeb(1,gteq(+energy,#float))?
:- modeb(1,gteq(+hydrophob,#float))?
:- modeb(1,lteq(+charge,#float))?
:- modeb(1,lteq(+energy,#float))?
:- modeb(1,lteq(+hydrophob,#float))?
:- modeb(1,(+charge)=(#charge))?
:- modeb(1,(+energy)=(#energy))?
:- modeb(1,(+hydrophob)=(#hydrophob))?
:- modeb(*,benzene(+drug,-ring))?
:- modeb(*,carbon_5_aromatic_ring(+drug,-ring))?
:- modeb(*,carbon_6_ring(+drug,-ring))?
:- modeb(*,hetero_aromatic_6_ring(+drug,-ring))?
:- modeb(*,hetero_aromatic_5_ring(+drug,-ring))?
:- modeb(*,ring_size_6(+drug,-ring))?
:- modeb(*,ring_size_5(+drug,-ring))?
:- modeb(*,nitro(+drug,-ring))?
:- modeb(*,methyl(+drug,-ring))?
:- modeb(*,anthracene(+drug,-ringlist))?
:- modeb(*,phenanthrene(+drug,-ringlist))?
:- modeb(*,ball3(+drug,-ringlist))?
:- modeb(*,member(-ring,+ringlist))?
:- modeb(1,member(+ring,+ringlist))?
:- modeb(1,connected(+ring,+ring))?

:- determination(active/1,atm/5)?
:- determination(active/1,bond/4)?
:- determination(active/1,gteq/2)?
:- determination(active/1,lteq/2)?
:- determination(active/1,'='/2)?
:- determination(active/1,lumo/2)?
:- determination(active/1,logp/2)?
:- determination(active/1,benzene/2)?
:- determination(active/1,carbon_5_aromatic_ring/2)?
:- determination(active/1,carbon_6_ring/2)?
:- determination(active/1,hetero_aromatic_6_ring/2)?
:- determination(active/1,hetero_aromatic_5_ring/2)?
:- determination(active/1,ring_size_6/2)?
:- determination(active/1,ring_size_5/2)?
:- determination(active/1,nitro/2)?
:- determination(active/1,methyl/2)?
:- determination(active/1,anthracene/2)?
:- determination(active/1,phenanthrene/2)?
:- determination(active/1,ball3/2)?
:- determination(active/1,member/2)?
:- determination(active/1,connected/2)?
```

B.3.5 Mesh Design

```
:-set(noise,1)?
:-set(nodes,7900)?
:-set(verbose,1)?

:-modeh(1, mesh(+edge,#int))?

:-modeb(1,long(+edge))?
:-modeb(1,usual(+edge))?
:-modeb(1,short(+edge))?
:-modeb(1,circuit(+edge))?
:-modeb(1,half_circuit(+edge))?
:-modeb(1,quarter_circuit(+edge))?
:-modeb(1,short_for_hole(+edge))?
:-modeb(1,long_for_hole(+edge))?
:-modeb(1,circuit_hole(+edge))?
:-modeb(1,half_circuit_hole(+edge))?
:-modeb(1,not_important(+edge))?
:-modeb(1,free(+edge))?
:-modeb(1,one_side_fixed(+edge))?
:-modeb(1,two_side_fixed(+edge))?
:-modeb(1,fixed(+edge))?
:-modeb(1,not_loaded(+edge))?
:-modeb(1,one_side_loaded(+edge))?
```

```
:-modeb(1,two_side_loaded(+edge))?
:-modeb(1,cont_loaded(+edge))?
:-modeb(6,neighbour(+edge,-edge))?
:-modeb(6,opposite(+edge,-edge))?
```

B.3.6 Traffic Problem Detection

```
:- set(verbose,2)?
:- set(nodes,1100)?

:- modeh(1,accident(+section,+time))?
:- modeh(1,congestion(+section,+time))?
:- modeh(1,noncs(+section,+time))?
:- modeb(1,velocidadd(+time,+section,#velocity))?
:- modeb(1,ocupaciond(+time,+section,#velocity))?
:- modeb(1,saturaciond(+time,+section,#velocity))?
:- modeb(*,secciones_posteriores(+section,-section))?
:- modeb(*,secciones_posteriores(-section,+section))?
:- modeb(1,tipo(+section,#stype))?
```

B.3.7 Minesweeper

```
:-set(i,5)?
:-set(verbose,2)?
:-set(c,8)?
:-set(r,50000)?
:-set(h,10000)?
:-set(nodes,4000)?
:-set(inflate,600)?

:-modeh(1, safe(+tileUK,+board))?
:-modeb(1, zoneOfInterest(+tileUK, +board, -zone))?
:-modeb(1, totalMinesLeft(+board,-nat))?
:-modeb(1, allMinesInFringe(+board, -setP))?
:-modeb(24, setHasXMines(-tileK, +board, +zone, -setP))?
:-modeb(1, diffSetHasXMines(+setP, +setP, -setP))?
:-modeb(1, unknownNotInSet(+tileUK, +board, +setP))?
:-modeb(1, inSet(+tileUK, +setP))?
:-modeb(1, lengthSet(+setP,+nat))?
:-modeb(1, minesInSet(+setP,-nat))?
:-modeb(1, minesInSet(+setP,#nat))?
```

Bibliography

[1] http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[3] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[4] Hendrik Blockeel. *Top-down induction of first order logical decision trees.* PhD thesis, Katholieke Universiteit Leuven, 1998.

[5] Hendrik Blockeel and Luc De Raedt. Lookahead and discretization in ILP. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th Int. Workshop on ILP*, volume 1297 of *Lecture Notes in AI*, pages 77–84, 1997.

[6] Hendrik Blockeel and Luc De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[7] Hendrik Blockeel, Sašo Džeroski, and Jasna Grbovic. Simultaneous prediction of multiple chemical parameters of river water quality with TILDE. In J. Zytkow and J. Rauch, editors, *Proc. of the 3rd European Conf. on Principles of Data Mining and Knowledge Discovery*, volume 1704 of *Lecture Notes in AI*, pages 32–40, 1999.

[8] Ivan Bratko, Stephen. Muggleton, and Aram Karalič. Applications of Inductive Logic Programming. In R.S. Michalski, I. Bratko, and M. Kubat, editors, *Machine Learning and Data Mining: methods and applications.* John Wiley and Sons Ltd., 1998.

[9] Michael Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1–2):85–99, 2002.

[10] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. Correction, ibidem 1(3):101-102.

[11] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.

[12] William W. Cohen. Rapid prototyping of ILP systems using explicit bias. In *Proc. of the IJCAI Workshop on ILP*, pages 24–35, 1993.

[13] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[14] William W. Cohen. Pac-learning non-recursive Prolog clauses. *Artificial Intelligence*, 79(1):1–38, 1995.

[15] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.

[16] Luc de Raedt. Logical settings from concept learning. *Artificial Intelligence*, 95(1):187–201, 1997.

[17] Luc De Raedt. Attribute value learning versus inductive logic programming: The missing links (extended abstract). In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 1–8, 1998.

[18] Luc De Raedt and Maurice Bruynooghe. An overview of the interactive concept-learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–192. Academic Press, 1992.

[19] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In S. Muggleton, editor, *Proc. of the 3rd Int. Workshop on ILP*, pages 25–40, 1993.

[20] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2–3):99–146, 1997.

[21] Luc De Raedt and Wim Van Laer. Inductive constraint logic. In *Proc. of the 6th Conf. on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in AI*, pages 80–94. Springer-Verlag, 1995. http://www.cs.kuleuven.ac.be/~wimv/ICL/main.html.

[22] Luc Dehaspe and Luc De Raedt. Parallel inductive logic programming. In *Proc. of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.

[23] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, 2001.

[24] Thomas G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857, 2000.

[25] Bojan Dolšak, Ivan Bratko, and Anton Jezernik. Application of machine learning in finite element computation. In R.S. Michalski, I. Bratko, and M. Kubat, editors, *Machine learning and data mining: methods and applications*, pages 147–171. John Wiley and Sons Ltd., 1998.

[26] Bojan Dolšak, Anton Jezernik, and Ivan Bratko. A knowledge base for finite element mesh design. *Artificial Intelligence in Engineering*, 9(1):19–27, 1994.

[27] Inês C. Dutra, C. David Page, Vitor Santos Costa, and Jude Shavlik. An empirical evaluation of bagging in ILP. In S. Matwin and C. Sammut, editors, *Proc. of the 12th Int. Conf. on ILP*, volume 2583 of *Lecture Notes in AI*, pages 48–65, 2003.

[28] Sašo Džeroski. Relational data mining applications: an overview. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 339–364. Springer-Verlag, 2001.

[29] Sašo Džeroski and Ivan Bratko. Handling noise in inductive logic programming. In S. Muggleton, editor, *Proc. of the 2nd Int. Workshop on ILP*, Report ICOT TM-1182, 1992.

[30] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 11–22, 1998.

[31] Sašo Džeroski and Tomaž Erjavec. Induction of Slovene nominal paradigms. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th Int. Workshop on ILP*, volume 1297 of *Lecture Notes in AI*, pages 141–148, 1997.

[32] Sašo Džeroski, Nico Jacobs, Martin Molina, Carlos Moure, Stephen Muggleton, and Wim Van Laer. Detecting traffic problems with ILP. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 281–290, 1998.

[33] Sašo Džeroski and Nada Lavrač. Introduction to inductive logic programming. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 48–73. Springer-Verlag, 2001.

[34] Werner Emde and Dietrich Wettschereck. Relational instance-based learning. In L. Saitta, editor, *Proc. of the 13th. ICML*, pages 122–130, 1996.

[35] Peter A. Flach. Knowledge representation for inductive learning. In A. Hunter and S. Parsons, editors, *Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU)*, volume 1638 of *Lecture Notes in AI*, pages 160–167, 1999.

[36] Peter A. Flach, Christophe Giraud-Carrier, and John W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 185–194, 1998.

[37] Peter A. Flach and Nicolas Lachiche. 1BC: A first-order Bayesian classifier. In S. Džeroski and P. A. Flach, editors, *Proc. of the 9th Int. Workshop on ILP*, volume 1634 of *Lecture Notes in AI*, pages 92–103, 1999.

[38] Hiroshi Fujita, Naoki Yagi, Tomonobu Ozaki, and Koichi Furukawa. A new design and implementation of Progol by bottom-up computation. In S. Muggleton, editor, *Inductive Logic Programming*, volume 1314 of *Lecture Notes in AI*, pages 163–174, 1996.

[39] Johannes Fürnkranz. Integrative windowing. *Journal of Artificial Intelligence Research*, 8:129–164, 1998.

[40] Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, January 1999.

[41] Johannes Fürnkranz and Peter A. Flach. An analysis of rule evaluation metrics. In *Proc. of the 20th ICML*, pages 202–209, 2003.

[42] Koichi Furukawa, Tomoko Murakami, Ken Ueno, Tomonobu Ozaki, and Keiko Shimazu. On a sufficient condition for the existence of most specific hypothesis in Progol. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th Int. Conf. on ILP*, volume 1297 of *Lecture Notes in AI*, pages 157–164, 1997.

[43] James Graham, C. David Page, and Alan Wild. Parallel data mining for pharmacophore discovery. In *Proc. of the Int. Conf. on Systems, Man and Cybernetics*, pages 1894–1899, 2000.

[44] Robert C. Holte, Liane E. Acker, and Bruce W. Porter. Concept learning and the problem of small disjuncts. In *Proc. of the 11th IJCAI*, pages 813–818, 1989.

[45] Tamás Horváth, Zoltán Alexin, Tibor Gyimóthy, and Stefan Wrobel. Application of different learning methods to Hungarian part-of-speech tagging. In S. Džeroski and P. A. Flach, editors, *Proc. of the 9th Int. Workshop on ILP*, volume 1634 of *Lecture Notes in AI*, pages 128–139, 1999.

[46] Tamás Horváth, Stefan Wrobel, and Uta Bohnebeck. Relational instance-based learning with list and terms. *Machine Learning*, 43(1/2):53–80, 2001.

[47] Andreas Junghanns. *Pushing the limits: new developments in single-agent search.* PhD thesis, University of Alberta, 1999.

[48] Aram Karalič and Ivan Bratko. First order regression. *Machine Learning*, 26(2/3):147–176, 1997.

[49] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, Spring 2000.

[50] Kristian Kersting and Luc De Raedt. Towards combining inductive logic programming with bayesian networks. In C. Rouveirol and M. Sebag, editors, *Proc. of the 11th Int. Conf. on ILP*, volume 2157 of *Lecture Notes in AI*, pages 118–131. Springer-Verlag, 2001.

[51] Jörg-Uwe Kietz and Sašo Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.

[52] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*, pages 335–359. Academic Press, 1992.

[53] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[54] Richard E. Korf. Iterative-deepening A*: An optimal admissible tree search. In *Proc. of the 9th IJCAI*, pages 1034–1036, 1985.

[55] Richard E. Korf. Macro-Operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[56] Stefan Kramer, Nada Lavrač, and Peter A. Flach. Propositionalization approaches to relational data mining. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 262–291. Springer-Verlag, 2001.

[57] Nada Lavrač, Sašo Džeroski, and Marko Grobelnik. Learning nonrecursive definitions of relations with LINUS. In Y. Kodratoff, editor, *Proc. of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in AI*, pages 265–281, 1991.

[58] Nada Lavrač and Peter A. Flach. An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic*, 2(4):458–494, October 2001.

[59] Nada Lavrač, Peter A. Flach, and Blaz Zupan. Rule evaluation measures: a unifying view. In S. Džeroski and P. A. Flach, editors, *Proc. of the 9th Int. Workshop on ILP*, volume 1634 of *Lecture Notes in AI*, pages 174–185, 1999.

[60] Nada Lavrač, Filip Železný, and Peter A. Flach. RSD: Relational subgroup discovery through first-order feature construction. In S. Matwin and C. Sammut, editors, *Proc. of the 12th Int. Conf. on ILP*, volume 2583 of *Lecture Notes in AI*, pages 149–165, 2003.

[61] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[62] Suresh Manandhar, Sašo Džeroski, and Tomaž Erjavec. Learning multilingual morphology with CLOG. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 135–144, 1998.

[63] Tohgoroh Matsui, Nobuhiro Inuzuka, Hirohisa Seki, and Hidenori Itoh. Comparison of three parallel implementations of an induction algorithm. In *Proc. of 8th Int. Parallel Computing Workshop*, pages 181–188, 1998.

[64] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[65] Tom M. Mitchell. *Machine Learning.* McGraw Hill, first edition, 1997.

[66] Raymond J. Mooney and Mary Elaine Califf. Induction of first-order decision lists: Results on learning the past tense of english verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995. http://www.cs.utexas.edu/users/ml/foidl.html.

[67] Eduardo Morales. Learning playing strategies in chess. *Computational Intelligence*, 12(1):65–87, 1996.

[68] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[69] Stephen Muggleton. Completing inverse entailment. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 245–249, 1998.

[70] Stephen Muggleton and Wray L. Buntine. Machine invention of first-order predicates by inverting resolution. In J. Laird, editor, *Proc. of the 5th Conf. on Machine Learning*, pages 339–352, 1988.

[71] Stephen Muggleton and Luc de Raedt. Inductive logic programming: theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

[72] Stephen Muggleton and Cao Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, volume 38 of *APIC Series*, pages 281–298. Academic Press, 1992.

[73] Stephen Muggleton and John Firth. Relational rule induction with CProgol4.4: a tutorial introduction. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 160–187. Springer-Verlag, 2001.

[74] Stephen Muggleton, Ross D. King, and Michael J. E. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5:647–657, 1992.

[75] Tomofumi Nakano, Nobuhiro Inuzuka, Hirohisa Seki, and Hidenori Itoh. Inducing Shogi heuristics using inductive logic programming. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, pages 155–164, 1998.

[76] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in AI.* Springer-Verlag, 1997.

[77] Michael J. Pazzani and Clifford A. Brunk. Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning. *Knowledge Acquisition*, 3:157–173, 1991.

[78] Michael J. Pazzani, Clifford A. Brunk, and Glenn Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proc. of the 8th Int. Workshop on Machine Learning*, pages 432–436, 1991.

[79] Lourdes Peña Castillo and Stefan Wrobel. Macro-operators in multirelational learning: a search-space reduction technique. In T. Elomaa, H. Mannila, and H. T. T. Toivonen, editors, *Proc. of the 13th ECML*, volume 2430 of *Lecture Notes in AI*, pages 357– 368, 2002.

[80] Lourdes Peña Castillo and Stefan Wrobel. Multirelational active learning for games. In G. Kokai and J. Zeidler, editors, *Machine Learning Workshop FGML 2002 (Treffen der GI-Fachgruppe 1.1.3 Maschinelles Lernen)*, pages 108–112, 2002.

[81] Lourdes Peña Castillo and Stefan Wrobel. On the stability of example-driven learning systems: a case study in multirelational learning. In C. A. Coello Coello, A. de Albornoz, E. Sucar, and O. Cairo, editors, *Proc. of the 2nd MICAI*, volume 2313 of *Lecture Notes in AI*, pages 321–330, 2002.

[82] Lourdes Peña Castillo and Stefan Wrobel. Learning minesweeper with multirelational learning. In *Proc. of the 18th IJCAI*, pages 533–538, 2003.

[83] Lourdes Peña Castillo and Stefan Wrobel. A comparative study on methods for reducing myopia of hill-climbing search in multirelational learning. In *Proc. of the 21st ICML*, 2004.

[84] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[85] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[86] J. Ross Quinlan. Determinate literals in inductive logic programming. In J. Mylopoulos and R. Reiter, editors, *Proc. of the 12th IJCAI*, volume 2, pages 746–750, 1991.

[87] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[88] J. Ross Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

[89] J. Ross Quinlan and R. Michael Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Computing*, 13(3-4):287–312, 1995.

[90] J. Ross Quinlan and R. Michael Cameron-Jones. Oversearching and layered search in empirical learning. In *Proc. of the 14th IJCAI*, volume 2, pages 1019–1024, 1995.

[91] Jan Ramon, Tom Francis, and Hendrik Blockeel. Learning a tsume-go heuristic with TILDE. In T. A. Marsland and I. Frank, editors, *Proc. of the 2nd Int. Conf. Computers and Games*, volume 2063 of *Lecture Notes in AI*, pages 151–169, 2001.

[92] Jan Ramon, Nico Jacobs, and Hendrik Blockeel. Opponent modeling by analysing play. In M. Bowling, G. Kaminka, and R. Vincent, editors, *Proc. of workshop on agents in computer games*, 2002.

[93] John D. Ramsdell, November 25, 2002. Personal communication.

[94] Bradley L. Richards and Raymond J. Mooney. Learning relations by pathfinding. In *Proc. of the 10th AAAI*, pages 50–55, 1992.

[95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, first edition, 1995.

[96] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[97] Ehud Y. Shapiro. An algorithm that infers theories from facts. In *Proc. of the 7th IJCAI*, pages 446–451, 1981.

[98] Glenn Silverstein and Michael J. Pazzani. Relational clichés: constraining constructive induction during relational learning. In L. Birnbaum and G. Collins, editors, *Proc. of the 8th Int. Workshop on Machine Learning*, pages 203–207, 1991.

[99] Ashwin Srinivasan, Stephen Muggleton, and Michael Bain. The justification of logical theories. In S. Muggleton, D. Michie, and K.Furukawa, editors, *Machine Intelligence*, volume 13, pages 87–121. Oxford University Press, 1994.

[100] Ashwin Srinivasan, Stephen Muggleton, Ross D. King, and Michael J. E. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1–2):277–299, 1996.

[101] Gerald Tesauro. Temporal–difference learning and td–gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[102] Ljupčo Todorovski, Irene Weber, Nada Lavrač, Olga Stěpánkova, Sašo Džeroski, Dimitar Kazakov, Darko Zupanič, and Peter Flach. Internet resources on ILP for KDD. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 375–388. Springer-Verlag, 2001.

[103] Stefan Wrobel. Inductive logic programming. In G. Brewka, editor, *Principles of Knowledge Representation.* CSLI, 1996.

[104] Stefan Wrobel. An algorithm for multi-relational discovery of subgroups. In J. Komorowski and J. Zytkow, editors, *Proc. of the 1st. European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 78–87. Springer-Verlag, 1997.

[105] Stefan Wrobel. Inductive logic programming for knowledge discovery in databases. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 48–73. Springer-Verlag, 2001.

[106] Akihiro Yamamoto. Improving Theories for Inductive Logic Programming Systems with Ground Reduced Programs. Technical Report AIDA–96–19, FG Intellektik, FB Informatik, TH Darmstadt, December 1996.

[107] Akihiro Yamamoto. Which hypotheses can be found with inverse entailment? In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th Int. Conf. on ILP*, volume 1297 of *Lecture Notes in AI*, pages 296–308, 1997.

[108] Jean-Daniel Zucker and Jean-Gabriel Ganascia. Learning structurally indeterminate clauses. In D. Page, editor, *Proc. of the 8th Int. Conf. on ILP*, volume 1446 of *Lecture Notes in AI*, pages 235–244, 1998.

Index