Inductive Logic Programming

Techniques and Applications

Nada Lavrač and Sašo Džeroski

Jožef Stefan Institute
Ljubljana, Slovenia

This book was published by Ellis Horwood, New York in 1994.
It is out of print and the copyright now resides with the authors.
Please reference this book as follows:

N. Lavrač and S. Džeroski.
Inductive Logic Programming: Techniques and Applications.
Ellis Horwood, New York, 1994.

Contents

IV Applications of ILP 197

Foreword

This book is about inductive logic programming (ILP), which is a new technology combining principles of inductive machine learning with the representation of logic programming. The new technology aims at inducing general rules starting from specific observations and background knowledge.

Let me explain why inductive logic programming is currently regarded as (one of) the most important recent development(s) in machine learning research. In my view, ILP = I $\cap$ LP, i.e., ILP is the intersection of techniques and interests in induction and logic programming. This makes inductive logic programming more powerful than traditional techniques that learn from examples, namely, inductive logic programming uses an expressive first-order logic framework instead of the traditional attribute-value framework, and facilitates the use of background knowledge. The first point is important because many domains of expertise cannot be formulated in an attribute-value framework. The second point is also significant because background knowledge is the key to success in nearly all applications of artificial intelligence. At the same time, inductive logic programming has room for both theory and practice. Inductive logic programming has a strong theoretical basis as it inherited many results from logic programming and computational learning theory, and adapted (and continues to adapt) these to fulfill the needs of a new discipline. Inductive logic programming has also very impressive applications in scientific discovery, knowledge synthesis and logic programming. To mention an important achievement: Stephen Muggleton has recently produced — using general purpose inductive logic programming systems — new scientific knowledge in drug design and protein engineering, which is now published in scientific journals [King et al. 1992, Muggleton et al. 1992a].

The authors of this book, Nada Lavrač and Sašo Džeroski, are members of the AI Laboratory of the Jožef Stefan Institute (Ljubljana, Slovenia) directed by Ivan Bratko. The laboratory and its members have an outstanding reputation in applying both machine learning and logic programming to real-life knowledge engineering problems. Nada Lavrač has already co-authored books in both areas: *Prolog through Examples: A Practical Programming Guide* (by Igor Kononenko and Nada Lavrač, Sigma Press, 1988) and *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems* (by Ivan Bratko, Igor Mozetič and Nada Lavrač, The MIT Press, 1989). It is therefore no surprise that Nada Lavrač, now with Sašo Džeroski, combines her two interests in this book.

The book is written from an engineering perspective with applications in mind. Indeed, after an introduction to and a survey of inductive logic programming in Part I, the authors focus on the empirical setting for inductive logic programming in Part II. The empirical setting is currently the one best suited for applications as it aims at synthesizing knowledge from potentially large sets of observations. In this second part, the authors present the first important contribution: the idea that inductive logic programming can benefit from the well-understood traditional induction techniques such as top-down induction of decision trees. This idea forms the basis of the system LINUS (and its extension DINUS) which inherits its efficiency from the traditional learning algorithms. The efficiency of LINUS is very relevant, not only to the users, but also to computational learning theory. Indeed, it was recently discovered that some of the assumptions of LINUS/DINUS are closely related to the class of polynomially learnable predicates in computation learning theory (PAC-learnability). Another significant contribution of LINUS is that it is the first inductive logic programming system able to handle numerical data. In Part III, the authors tackle another key problem when doing real-life applications: imperfect data. To this aim, they investigate and evaluate new heuristics for inductive logic programming, and show in this way that inductive logic programming is ready to handle imperfect data and therefore also real-life applications. Real-life applications are the subject of Part IV, which contains some impressive results in medicine (the award-winning application in rheumatology at the Third Scandinavian Conference on Artificial Intelligence, SCAI-91), in engineering (finite element mesh design), and qualitative modeling.

Besides giving a good introduction to the field of inductive logic programming, this book also shows what (knowledge) engineers can achieve with this new technology. Furthermore, it is easily readable and accessible. Therefore, I believe it will be of interest to a wide audience, including graduates, researchers and practitioners of artificial intelligence and computer science; in particular, those interested in machine learning, knowledge engineering, knowledge discovery in databases and logic programming.

Sint-Amandsberg, April 1993 *Luc De Raedt*

Preface

Inductive logic programming (ILP) is a research area at the intersection of machine learning and logic programming. It aims at a formal framework as well as practical algorithms for inductive learning of relational descriptions in the form of logic programs. From logic programming, ILP has inherited its sound theoretical basis, and from machine learning, an experimental approach and orientation towards practical applications. ILP has already shown its application potential in the following areas: knowledge acquisition, inductive program synthesis, inductive data engineering, and knowledge discovery in databases.

This book is an introduction to this exciting new field of research. It should be of interest to the following readership: knowledge engineers concerned with the automatic synthesis of knowledge bases for expert systems; software engineers who could profit from inductive programming tools; researchers in system development and database methodology, interested in techniques for knowledge discovery in databases and inductive data engineering; and researchers and graduate students in artificial intelligence, machine learning, logic programming, software engineering and database methodology.

As ILP is a relatively young field, the ILP literature consists mainly of conference and workshop proceedings. A book collection of papers [Muggleton 1992a], covering the whole field of ILP, is also available, as well as a book on theory revision in the context of interactive ILP [De Raedt 1992]. Interactive ILP, one of the two major subfields of ILP, is closely related to program debugging and theory revision where a small number of examples is available. The second major subfield of ILP, called empirical ILP, places an emphasis on the extraction of regularities from a large number of possibly erroneous examples. The spirit of empirical ILP is much closer to the existing successful learning systems, and is therefore more suitable for practical applications than interactive ILP. This book extensively covers empirical ILP techniques and applications.

The book is divided into four parts. Part I is an introduction to the field of ILP. Part II describes in detail empirical ILP techniques and systems. Part III is concerned with the problem of handling imperfect data in ILP, whereas Part IV gives an overview of empirical ILP applications.

Part I comprises Chapters 1 to 3. Chapter 1 touches upon the topics covered by the book. Chapter 2 introduces the basic concepts and terminology of logic programming, deductive databases and inductive logic programming. The basic generalization and specialization techniques used in ILP are described in Chapter 3.

Chapters 4 to 7 constitute Part II. Chapter 4 gives an overview of several empirical ILP systems with an emphasis on FOIL [Quinlan 1990], which has been the basis for much of our work. The central chapter of this part, Chapter 5, gives a detailed description of the ILP system LINUS. This includes a description of the LINUS environment, the propositional learning systems incorporated into LINUS, the transformation algorithm, a specification of the hypothesis language and a complexity analysis. It is also shown how LINUS can be extended to generate hypotheses in a more expressive hypothesis language. Chapter 6 describes several experiments in learning relations with LINUS. In Chapter 7, the framework of refinement graphs is used to compare the hypothesis languages and the search complexity of LINUS and FOIL.

Part III starts with Chapter 8, which first describes the problem of handling imperfect data in attribute-value and ILP systems and then gives an overview of techniques and heuristics used in handling imperfect data. Chapter 9 presents the ILP system mFOIL, which incorporates techniques for handling imperfect data from attribute-value systems into the FOIL framework. Its language bias (search space) and search bias (heuristics, search strategy and stopping criteria) are described in detail. An experimental comparison of LINUS, FOIL and mFOIL in a chess endgame domain with artificially introduced errors (noise) in the examples is made in Chapter 10.

Although experiments in artificial domains under controlled amounts of noise reveal important performance aspects of ILP systems, the ultimate test is their application to real-life problems. Part IV gives

a detailed description of the applications of LINUS and mFOIL to three different practical learning problems. Comparisons are also made with FOIL and GOLEM [Muggleton and Feng 1990]. Chapter 11 describes the application of LINUS to the problem of learning rules for medical diagnosis, where examples and specialist background knowledge were provided by a medical expert. Chapter 12 compares the performance of mFOIL, FOIL and GOLEM on the task of inducing rules for determining the appropriate resolution of a finite element mesh. Chapter 13 describes the induction of qualitative models from example behaviors and describes the application of mFOIL to this problem, making also a comparison to other systems. Finally, Chapter 14 concludes with an overview of several applications of GOLEM to illustrate the potential of ILP for practical applications.

Acknowledgements

The book is a result of several years of research. Most of it was done at the Artificial Intelligence Laboratory of the Computer Science Department, Jožef Stefan Institute in Ljubljana. We would like to thank Ivan Bratko, the head of the Artificial Intelligence Laboratory, for guiding our research interests towards challenging research topics in AI. Our thanks goes also to Marjan Špegel, the head of the Computer Science Department, for his support during all these years of research.

Our research was based on the tradition of the Artificial Intelligence Laboratory in the areas of machine learning and qualitative modeling. The KARDIO project [Bratko et al. 1989] has been a motivation and a source of ideas for our work. The special purpose learning system QuMAS [Mozetič 1987], which was used in KARDIO for reconstructing a qualitative model of the heart, was based on the idea of transforming a relation learning problem into propositional form. This idea has been further developed into our general ILP environment called LINUS [Lavrač et al. 1991a]. This approach has provided us with a possibility to use a variety of learning techniques, including mechanisms for handling imperfect data, developed in propositional learning systems. Besides using the noise-handling techniques within the LINUS framework, we have also adapted them for direct use in our ILP learner mFOIL [Džeroski and Bratko 1992a], an extension of the FOIL [Quinlan 1990] system. LINUS and mFOIL transcend the approaches of QuMAS and

FOIL mainly by the use of more sophisticated mechanisms for handling imperfect data, which is one of the main topics of this book. We are grateful to Igor Mozetič for his support in the development of LINUS, and to Bojan Cestnik who contributed his expertise and advice in the selection of the noise-handling mechanisms described in this book.

We wish to thank Peter Flach, Igor Kononenko, Stephen Muggleton, Tanja Urbančič and, in particular, Luc De Raedt who contributed significantly to this book by providing helpful comments and advice. We would further like to thank Marko Grobelnik and Dunja Mladenič, who contributed to the development of LINUS and a VAX/VMS version of ASSISTANT incorporated into LINUS, respectively; Bruno Stiglic and Robert Trappl for their support and interest in this work; Ross Quinlan for making his FOIL system available for experimental comparisons; Michael Bain, Ivan Bratko, Bojan Dolšak, Vladimir Pirnat and Ross Quinlan for making their experimental data available; Bojan Orel for his $\TeX$pertise used in the preparation of this book; and our colleagues in the Artificial Intelligence Laboratory for their contributions to the views expressed in this book.

Our research has been continuously supported by the Slovenian Ministry of Science and Technology. The research was also part of the ESPRIT II Basic Research Action no. 3059 ECOLES and the ESPRIT III Basic Research Project no. 6020 Inductive Logic Programming. The Slovenian Ministry of Science and Technology supported Nada Lavrač during her three-month visit to the University of Illinois at Urbana-Champaign, Urbana, Illinois in 1985, and to the George Mason University, Fairfax, Virginia in 1988. The Belgian State-Science Policy Office grant enabled her a six-month stay at the Katholieke Universiteit Leuven, Belgium in 1991–1992. The Slovenian Ministry of Science and Technology and the British Council supported Sašo Džeroski during his eight-month stay at the Turing Institute, Glasgow, UK, also in 1991–1992. We are grateful to Ryszard Michalski (George Mason University), to Maurice Bruynooghe and Luc De Raedt (Katholieke Universiteit Leuven), and to Stephen Muggleton (The Turing Institute) for their support of our work in the stimulating research environments of their institutions.

The inductive logic programming community, gathered in the ESPRIT III project Inductive Logic Programming, provided a stim-

ulating environment for the exchange of ideas and results in the field; this book is a contribution to the body of knowledge developed within this community. We are grateful to Stephen Muggleton and Luc De Raedt for their contributions to ILP research and for their organizational efforts which have made this research group so lively.

Copyright acknowledgements

The following material was taken from the book *Inductive Logic Programming*, edited by Stephen Muggleton, with the kind permission of Academic Press: Figure 5.1, Figure 6.6, Sections 7.4, 7.5, 7.6 and Figure 14.9.

A major part of Chapter 11 appears in *Applied Artificial Intelligence* 7: 273–293, 1993, under the title 'The utility of background knowledge in learning medical diagnostic rules'. It is republished with the permission of the publisher Taylor & Francis.

Thanks to Bojan Dolšak for Figure 12.1, Dunja Mladenić and Aram Karalič for Figure 14.3, and Ashwin Srinivasan for Figure 14.1.

Ljubljana, March 1993 *Nada Lavrač* and *Sašo Džeroski*

Part I

Introduction to ILP

1

Introduction

Inductive logic programming is a research area at the intersection of machine learning and logic programming. It aims at a formal framework and practical algorithms for inductively learning logic programs from examples. This chapter gives an introduction to inductive logic programming. It first acquaints the reader with the basics of inductive concept learning and discusses the role of background knowledge and language bias. It then gives a brief overview of inductive logic programming, addresses the problem of handling imperfect data and discusses the potential of inductive logic programming for practical applications.

1.1 Inductive concept learning

The current interest in *machine learning*, a subfield of artificial intelligence, is twofold. On the one hand, although there is no consensus on the nature of intelligence, it is agreed that the capability of learning is crucial for any intelligent behavior. Machine learning is thus justified from the scientific point of view as part of cognitive science. On the other hand, machine learning techniques can be successfully used in knowledge acquisition tools, which is a justification from the engineering point of view. Namely, it is generally accepted that the main problem in building expert systems (to which the success of artificial intelligence in industry is largely due) is the acquisition of knowledge.

As part of computer science/engineering, the goal of machine learning is to develop methods, techniques and tools for building intelligent *learning machines*, e.g., such learning programs which are able to change themselves in order to perform *better* at a given task. To perform 'better' means, for example, to perform more efficiently and/or more accurately. 'Better' can also denote the learner's ability to handle a broader class of problems.

In a broad sense, machine learning paradigms [Carbonell 1989] include inductive learning, analytic (deductive) learning, learning with genetic algorithms, and connectionist learning (learning with neural nets). Furthermore, several learning paradigms can be integrated within a single multistrategy learning system [Michalski and Tecuci 1991]. In a narrower sense, Michie's strong criterion [Michie 1988] defines learning as the ability to acquire new knowledge by requiring the result of learning to be understandable by humans. In this sense, connectionist systems are not considered to be learning systems.

This book focuses on *inductive learning*. It is generally known that induction means reasoning from specific to general. In the case of *inductive learning from examples*, the learner is given some examples from which general rules or a theory underlying the examples are derived. Inductive learning has been successfully applied to a variety of classification and prediction problems, such as the diagnosis of a patient or a plant disease, or the prediction of mechanical properties of steel on the basis of its chemical characteristics. These problems can be formulated as tasks of learning concepts from examples, referred to as *inductive concept learning*, where classification rules for a specific concept must be induced from instances (and non-instances) of that concept.

To define the problem of inductive concept learning one first needs to define a *concept*. If $\mathcal{U}$ is a universal set of objects (or observations), a concept $\mathcal{C}$ can be formalized as a subset of objects in $\mathcal{U}$: $\mathcal{C} \subseteq \mathcal{U}$. For example, $\mathcal{U}$ may be the set of all patients in a register, and $\mathcal{C} \subseteq \mathcal{U}$ the set of all patients having a particular disease. Similarly, the concept 'arch' is the set of all arches in some universe of objects $\mathcal{U}$. To learn a concept $\mathcal{C}$ means to learn to recognize objects in $\mathcal{C}$ [Bratko 1989], i.e., to be able to tell whether $x \in \mathcal{C}$ for each $x \in \mathcal{U}$.

Describing objects and concepts

In machine learning, a formal language for describing objects and concepts needs to be selected. Objects are described in an *object description language*. Concepts can be described in the same language or in a separate *concept description language*.

In an *attribute-value* object description language objects are described by a fixed repertoire of features, also called *attributes*, each of

them taking a value from a corresponding prespecified value set. In the game of poker, for instance, objects are poker cards and the selected two attributes for describing cards are the suit and the rank of a card. The set of values for the attribute $Suit$ is {*hearts, spades, clubs, diamonds*}, and the set of values for $Rank$ is {*7, 8, 9, 10, j, q, k, a* }, where *j*, *q*, *k*, *a* stand for *jack*, *queen*, *king* and *ace*, respectively.

An individual card can be described in an attribute-value language by a conjunctive expression, e.g., $[Suit = diamonds] \wedge [Rank = 7]$. In a different attribute-value language, this same object can be described by a tuple $< diamonds, 7 >$. In the first-order language of *Horn clauses*, used in logic programming, a card can be described by a *ground fact*, e.g., $card(diamonds, 7)$. In this case, *card* is a predicate name, and the values of the two arguments $Rank$ and $Suit$ of the predicate $card/2$ are the constants *diamonds* and 7.

A pair of cards can be described in an object description language comprising four attributes: $Suit_1$ and $Rank_1$ describing one card, and $Suit_2$ and $Rank_2$ describing the other card. In an attribute-value language, a pair of two cards $< diamonds, 7 >$ and $< hearts, 7 >$ can either be described by a 4-tuple $< diamonds, 7, hearts, 7 >$ or, alternatively, by a conjunctive expression $[Suit_1 = diamonds] \wedge [Rank_1 = 7] \wedge [Suit_2 = hearts] \wedge [Rank_2 = 7]$. In the language of Horn clauses, two alternative descriptions are $pair(diamonds, 7, hearts, 7)$ and $pair(card(diamonds, 7), card(hearts, 7))$, where *pair* is a predicate name and *card* is a function name.

Concepts can be described extensionally or intensionally. A concept is described *extensionally* by listing the descriptions of all of its instances. For example, the concept *pair* in poker can be extensionally described by the set of all pairs of cards which have the same rank, e.g., by the following set of 4-tuples: $pair = \{<hearts, 7, spades, 7>, < hearts, 7, clubs, 7 >, \ldots, < diamonds, a, clubs, a >\}$.

Extensional concept descriptions can be undesirable for a number of reasons, including the fact that a concept may contain an infinite number of instances. Consequently, it is preferable to describe concepts *intensionally*, that is, in a separate concept description language which allows for more compact and concise concept descriptions, e.g., in the form of rules. The following rule is, for example, an intensional description of the concept *pair* in an attribute-value concept description

language:

$$\begin{array}{lll} pair & \textbf{if} & [Rank_1 = 7] \wedge [Rank_2 = 7] \vee \\ & & [Rank_1 = 8] \wedge [Rank_2 = 8] \vee \\ & & \ldots \\ & & [Rank_1 = a] \wedge [Rank_2 = a] \end{array}$$

In this attribute-value language, the antecedent of an *if-then* rule is a disjunction of conditions, each condition being a conjunction of attribute-value pairs (expressions of the form $Attribute = value$). As the attributes $Suit_1$ and $Suit_2$ do not appear in the above rule, they can have any value in the appropriate range.

In case the concept description language is more powerful and allows for the use of variables, the concept *pair* can be described in a more compact way:

$$pair \quad \textbf{if} \quad Rank_1 = Rank_2$$

In the language of Horn clauses, the above concept description takes the following form:

$$pair(Suit_1, Rank_1, Suit_2, Rank_2) \quad \leftarrow \quad Rank_1 = Rank_2$$

If the description language allows for the use of function symbols for describing structured terms, the *atom* $pair(Suit_1, Rank_1, Suit_2, Rank_2)$ in the above clause could be replaced by the atom with structured terms in the arguments $pair(card(Suit_1, Rank_1), card(Suit_2, Rank_2))$, yielding:

$$pair(card(Suit_1, Rank_1), card(Suit_2, Rank_2)) \leftarrow Rank_1 = Rank_2$$

Definition of inductive concept learning

Having selected description languages for objects and concepts, a procedure is needed that will establish whether a given object belongs to a given concept, i.e., whether the description of the object *satisfies* the description of the concept. If it does, we say that the concept description *covers* the object description, or that the object description *is covered* by the concept description.

Let us, from now on, use the term *fact* for an object description and *hypothesis* for an intensional concept description to be learned. An

example e for learning a concept $\mathcal{C}$ is a labeled fact, with label $\oplus$, if the object is an instance of the concept $\mathcal{C}$, and label $\ominus$ otherwise (i.e., if it is a non-instance of the concept). Let $\mathcal{E}$ denote a set of examples. The facts from $\mathcal{E}$ labeled $\oplus$ form a set of *positive examples* $\mathcal{E}^+$ and those labeled $\ominus$ constitute a set of *negative examples* $\mathcal{E}^-$ for the concept $\mathcal{C}$. We will take the liberty of assuming that facts in $\mathcal{E}^+$ and $\mathcal{E}^-$ are not labeled explicitly, but rather implicitly by their membership in one of the two sets; as a consequence, we will sometimes use the term example instead of fact and vice versa. For example, when learning the concept *pair* in poker, $pair(card(diamonds, 7), card(spades, 7))$ is a positive example, and $pair(card(diamonds, 7), card(spades, 10))$ is a negative example.

In the so-called *single concept learning*, a concept description is induced from facts labeled $\oplus$ and $\ominus$, i.e., from examples and counterexamples of a single concept $\mathcal{C}$. In *multiple concept learning*, labels denote different concept names representing different *classes*. In this case, the set of training examples $\mathcal{E}$ can be divided into subsets of positive and negative examples for each individual concept (class).

The problem of learning a single concept $\mathcal{C}$ from examples can now be stated as follows:

Inductive concept learning Given a set $\mathcal{E}$ of positive and negative examples of a concept $\mathcal{C}$, find a hypothesis $\mathcal{H}$, expressed in a given concept description language $\mathcal{L}$, such that:

- every positive example $e \in \mathcal{E}^+$ is covered by $\mathcal{H}$,
- no negative example $e \in \mathcal{E}^-$ is covered by $\mathcal{H}$.

To test the coverage, a function

$$covers(\mathcal{H}, e) \tag{1.1}$$

can be introduced, which returns the value *true* if e is covered by $\mathcal{H}$, and *false* otherwise. The implementation of this function depends on the object and concept description languages. For instance, if e is

$$[Suit_1 = diamonds] \wedge [Rank_1 = 7] \wedge [Suit_2 = hearts] \wedge [Rank_2 = 7]$$

and $\mathcal{H}$ is

$$
\begin{array}{ll}
pair \ \textbf{if} & [Rank_1 = 7] \wedge [Rank_2 = 7] \bigvee \\
 & [Rank_1 = 8] \wedge [Rank_2 = 8] \bigvee \\
 & \ldots \\
 & [Rank_1 = a] \wedge [Rank_2 = a]
\end{array}
$$

the *covers* function simply tests whether e satisfies any of the disjuncts in $\mathcal{H}$. In logic programming, where a hypothesis is a set of program clauses and an example is a ground fact, a forward chaining procedure [Bry 1990] or a form of SLD-resolution rule of inference [Lloyd 1987] can be used to check whether e is entailed by $\mathcal{H}$. For example, $pair(card(diamonds, 7), card(hearts, 7))$ is entailed by

$$pair(card(Suit_1, Rank_1), card(Suit_2, Rank_2)) \leftarrow Rank_1 = Rank_2$$

The function $covers(\mathcal{H}, e)$ can be extended to work for sets of examples in the following way:

$$covers(\mathcal{H}, \mathcal{E}) = \{e \in \mathcal{E} \mid covers(\mathcal{H}, e) = true\} \qquad (1.2)$$

returning the set of facts from $\mathcal{E}$ which are covered by $\mathcal{H}$.

In the problem statement of inductive concept learning it is required that the hypothesis $\mathcal{H}$ covers all the positive examples and none of the negative ones. In this case we say that the hypothesis $\mathcal{H}$ is complete and consistent with respect to the examples $\mathcal{E}$. A hypothesis $\mathcal{H}$ is *complete* with respect to examples $\mathcal{E}$ if it covers all the positive examples, i.e., if $covers(\mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$. A hypothesis $\mathcal{H}$ is consistent with respect to examples $\mathcal{E}$ if it covers none of the negative examples, i.e., if $covers(\mathcal{H}, \mathcal{E}^-) = \emptyset$. Figure 1.1 shows different possible situations regarding the completeness/consistency of a hypothesis.

A similar formalization of the problem of concept learning is given in De Raedt [1992], where the completeness and consistency criteria are together referred to as a *quality criterion*.

Criteria of success

The above stated definition of inductive concept learning requires that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. There is, however, no a priori guarantee that $\mathcal{H}$ will correspond to $\mathcal{C}$ on other objects as well [Bratko 1989]. Since one of the major aims of learning is to classify unseen objects with respect to $\mathcal{C}$, one can view the induced hypothesis $\mathcal{H}$ also as a *classifier* of new objects. Therefore, the accuracy of classifying unseen objects is the main criterion of the success of the learning system.

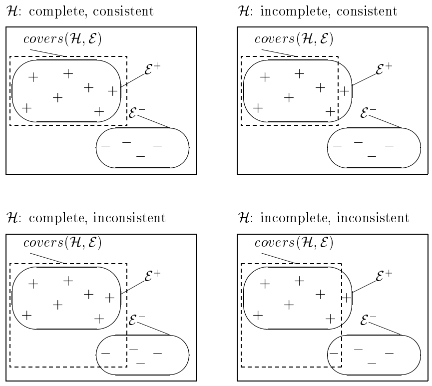

Figure 1.1: Completeness and consistency of a hypothesis.

Thus, important performance criteria of learning include the following:

Classification accuracy The classification accuracy of $\mathcal{H}$ is measured as the percentage of objects correctly classified by the hypothesis.

Transparency The transparency of $\mathcal{H}$ denotes the extent to which it is understandable to humans. A possible measure is the *length* of the hypothesis, expressed by the total number of conditions in an induced rule set $\mathcal{H}$. Another possibility is to measure the number of bits used in the encoding of the description.

Statistical significance Statistical significance tests can be used to evaluate whether hypothesis $\mathcal{H}$ represents a genuine regularity in the training examples and not a regularity which is due to chance.

Information content The information content or the relative information score of classifier $\mathcal{H}$ [Kononenko and Bratko 1991] scales the classifier's performance according to the difficulty of the classification problem. It is measured by taking into account the prior probability of the concept $\mathcal{C}$ to be learned, i.e., the percentage of objects in $\mathcal{U}$ that belong to $\mathcal{C}$. For example, in medical diagnosis a correct classification into a more probable diagnosis provides less information than a correct classification into a rare diagnosis, which is only represented by a few examples.

1.2 Background knowledge

Concept learning can be viewed as searching the space of concept descriptions [Mitchell 1982]. If the learner has no prior knowledge about the learning problem it learns exclusively from examples. However, difficult learning problems typically require a substantial body of prior knowledge. We will refer to declarative prior knowledge as *background knowledge*. Using background knowledge, the learner might express the generalization of examples in a more natural and concise manner.

The hypothesis language $\mathcal{L}$ and, indirectly, the background knowledge $\mathcal{B}$ determine the search space of possible concept descriptions, also called the *hypothesis space*. Therefore, the background knowledge has to be included in the problem statement of inductive concept learning.

Inductive concept learning with background knowledge Given a set of training examples $\mathcal{E}$ and background knowledge $\mathcal{B}$, find a hypothesis $\mathcal{H}$, expressed in some concept description language $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the background knowledge $\mathcal{B}$ and the examples $\mathcal{E}$.

To make this formulation of inductive concept learning operational, the function *covers* introduced in equations (1.1) and (1.2) must be extended to take into account the background knowledge $\mathcal{B}$:

$$covers(\mathcal{B}, \mathcal{H}, e) = covers(\mathcal{B} \cup \mathcal{H}, e) \qquad (1.3)$$

$$covers(\mathcal{B}, \mathcal{H}, \mathcal{E}) = covers(\mathcal{B} \cup \mathcal{H}, \mathcal{E}) \qquad (1.4)$$

The completeness and consistency requirements are also modified to read as stated below.

Completeness A hypothesis $\mathcal{H}$ is *complete* with respect to background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ if all the positive examples are covered, i.e., if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$.

Consistency A hypothesis $\mathcal{H}$ is consistent with respect to background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ if no negative example is covered, i.e., if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-) = \emptyset$.

1.3 Language bias

Any mechanism employed by a learning system to constrain the search for hypotheses is named *bias* [Utgoff and Mitchell 1982]. *Declarative bias* denotes explicit, user-specified bias which can preferably be formulated as a modifiable parameter of the system. Bias can either determine how the hypothesis space is searched (*search bias*) or determine the hypothesis space itself (*language bias*).

This section focuses on language bias. By selecting a stronger language bias (a less expressive hypothesis language) the search space becomes smaller and learning more efficient; however, this may prevent the system from finding a solution which is not contained in the less expressive language. This expressiveness/tractability tradeoff underlies much of the current research in inductive learning.

When considering language bias, the *expressiveness* of a hypothesis language becomes a crucial notion. The term *expressiveness* captures two different dimensions. The first dimension is the expressive power of a formalism; in its standard meaning, *stronger expressive power* means that there are concepts that can be represented in the stronger formalism which can not be represented in the weaker formalism. The other dimension is the *transparency* (or *length*) of the concept representation. For example, it is possible to state that two Boolean-valued attributes have the same value in both an attribute-value language and a first-order language. But, whereas in the former we have to say $([A = false] \wedge [B = false]) \vee ([A = true] \wedge [B = true])$, we can simply say $A = B$ in the latter.

Various logical formalisms have been used in inductive learning systems to represent examples and concept descriptions. These formalisms are similar to the formalisms for representing knowledge in general, ranging from propositional logic to full first-order predicate calculus. Within this scope one most frequently distinguishes between systems which learn attribute descriptions, and systems which learn first-order relational descriptions.

Several widely known inductive learning algorithms, such as ID3 [Quinlan 1986] and AQ [Michalski 1983, Michalski et al. 1986] use an attribute-value language to represent objects and concepts, and are thus called *attribute-value learners.* They are also called *propositional learners* since an attribute-value language is in its expressive power equivalent to propositional calculus. In both ID3 and AQ, objects are described in terms of their global features, i.e., by values of a fixed collection of *attributes.* To represent concepts, decision trees are used in ID3 and if-then rules in AQ. The two main limitations of propositional learners are the limited expressiveness of the representational formalism and their limited capability of taking into account the available background knowledge.

Another class of learning systems, called *relation learners*, induce descriptions of relations (definitions of predicates). In these systems, objects can be described structurally, i.e., in terms of their components and relations among the components. The given relations constitute the background knowledge. In relation learners, the languages used to represent examples, background knowledge and concept descriptions are typically subsets of first-order logic. In particular, the language of *logic programs* [Lloyd 1987] provides sufficient expressiveness for solving a significant number of relation learning problems. Learners that induce hypotheses in the form of logic programs are called *inductive logic programming* systems.

1.4 Inductive logic programming

Background knowledge plays a central role in relation learning, where the task is to define, from given examples, an unknown relation (i.e., the target predicate) in terms of (itself and) known relations from the background knowledge. If the hypothesis language of a relation learner is the language of logic programs, then learning is, in fact, logic program synthesis and has recently been named *inductive logic programming* (ILP) [Muggleton 1991, 1992a]. In ILP systems, the training examples, the background knowledge and the induced hypotheses are all expressed in a logic program form, with additional restrictions imposed on each of the three languages. For example, training examples are typically represented as ground facts of the target predicate, and most often background knowledge is restricted to be of the same form.

One of the early systems that made use of relational background knowledge in the process of learning structural descriptions was INDUCE [Michalski 1980]. The work on model inference by Shapiro [1983], Plotkin's work on inductive generalization [Plotkin 1969] and the work of Sammut and Banerji [1986] have inspired most of the current efforts in the area of inductive logic programming.

An example ILP problem

Let us illustrate the ILP task on a simple problem of learning family relations.

Example 1.1 (An ILP problem)
The task is to define the target relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations *female* and *parent*. These relations are given in Table 1.1. There are two positive and two negative examples of the target relation.

In the hypothesis language of Horn clauses it is possible to formulate the following definition of the target relation:

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

This definition is consistent and complete with respect to the background knowledge and the training examples. Depending on the background knowledge, the language $\mathcal{L}$ and the complexity of the target

Training examples		*Background knowledge*	
$daughter(mary, ann).$	$\oplus$	$parent(ann, mary).$	$female(ann).$
$daughter(eve, tom).$	$\oplus$	$parent(ann, tom).$	$female(mary).$
$daughter(tom, ann).$	$\ominus$	$parent(tom, eve).$	$female(eve).$
$daughter(eve, ann).$	$\ominus$	$parent(tom, ian).$	

Table 1.1: A simple ILP problem: learning the *daughter* relation.

concept, the target predicate definition may consist of a set of clauses, such as:

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$
$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

■

Dimensions of ILP

Inductive logic programming systems (as well as other inductive learning systems) can be divided along several dimensions. First of all, they can learn either a single concept or multiple concepts (predicates). Second, they may require all the training examples to be given before the learning process (batch learners) or may accept examples one by one (incremental learners). Third, during the learning process, a learner may rely on an oracle to verify the validity of generalizations and/or classify examples generated by the learner. The learner is called interactive in this case and non-interactive otherwise. Finally, a learner may try to learn a concept from scratch or can accept an initial hypothesis (theory) which is then revised in the learning process. The latter systems are called theory revisors.

Although these dimensions are in principle independent, existing ILP systems are situated at two ends of the spectrum. At one end are batch non-interactive systems that learn single predicates from scratch, while at the other are interactive and incremental theory revisors that learn multiple predicates. Following De Raedt [1992] we call the first

type of ILP systems *empirical ILP systems* and the second type *interactive ILP systems.* The second type of systems can alternatively be referred to as *incremental ILP systems.*

Interactive ILP systems include MIS [Shapiro 1983], MARVIN [Sammut and Banerji 1986], CLINT [De Raedt and Bruynooghe 1989, De Raedt and Bruynooghe 1992a] (described also in De Raedt [1992]), CIGOL [Muggleton and Buntine 1988] and other approaches based on inverting resolution [Rouveirol 1991, Wirth 1988, 1989]. Interactive ILP systems typically learn definitions of multiple predicates from a small set of examples and queries to the user. On the other hand, empirical ILP systems typically learn a definition of a single predicate from a large collection of examples. This class of ILP systems includes FOIL [Quinlan 1990], mFOIL [Džeroski 1991], GOLEM [Muggleton and Feng 1990], LINUS [Lavrač et al. 1991a], MARKUS [Grobelnik 1992], etc.

This book is mainly concerned with empirical ILP systems which already show their potential for practical applications. Part I of the book gives an overview of empirical ILP systems, with an emphasis on LINUS and FOIL.

1.5 Imperfect data

The definition of the inductive concept learning task from Section 1.1 requires that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. However, in practice it may happen that the object descriptions and their classifications (labels) presented to an inductive learning system contain various kinds of errors, either random or systematic. In such cases, we say that the learner has to deal with *imperfect data.* In this case, it may be advantageous not to insist that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. A very desirable property of an inductive learning system is then the ability to avoid the effects of imperfect data by distinguishing between genuine regularities in the examples and regularities due to chance or error.

When learning definitions of relations, the following kinds of data imperfection are usually encountered [Lavrač and Džeroski 1992b]:

- noise, i.e., random errors in the training examples and background knowledge,

- insufficiently covered example space, i.e., too sparse training examples from which it is difficult to reliably detect correlations,

- inexactness, i.e., inappropriate or insufficient description language which does not contain an exact description of the target concept, and
- missing values in the training examples.

Learning systems usually have a single mechanism for dealing with the first three types of imperfect data. Such mechanisms, often called *noise-handling mechanisms*, are typically designed to prevent the induced hypothesis $\mathcal{H}$ from *overfitting* the data set $\mathcal{E}$, i.e., to avoid concept descriptions $\mathcal{H}$ which agree with $\mathcal{C}$ perfectly on the training examples, but poorly on unseen examples. Accordingly, the completeness and consistency criteria in the definition of the inductive learning task need to be relaxed and replaced by a more general quality criterion which would allow the hypothesis to misclassify some of the training examples. Missing values are usually handled by a separate mechanism.

Successors of ID3 and AQ15, e.g., ASSISTANT [Cestnik et al. 1987], CN2 [Clark and Niblett 1989, Clark and Boswell 1991] and C4.5 [Quinlan 1993] have been successfully applied to a wide range of real-life domains and include several techniques for handling imperfect data. These include pruning (pre-pruning and post-pruning) of decision trees and the use of significance tests in rule truncation. Recently, improved probability estimates [Cestnik 1990] have been used to enhance the techniques used in tree pruning [Cestnik and Bratko 1991] and rule truncation [Clark and Boswell 1991, Džeroski et al. 1992a].

Until recently, relation learning systems had been mostly of interactive and incremental nature, relying on a small set of examples. The examples had been assumed correct, as well as the answers provided by the oracle. Thus, ILP systems did not address the issue of data imperfection. However, with the advent of empirical ILP systems, which reached the stage of efficiency where application to practical domains is feasible, the interest in handling imperfect data in ILP increased. Two recent relation learning systems, FOIL and LINUS, include sophisticated noise-handling mechanisms [Lavrač and Džeroski 1992b]. A post-processing mechanism based on reduced error pruning has also been used in FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], an

extension to FOIL that combines empirical and explanation-based learning. Bottom-up ILP systems (e.g., CIGOL and GOLEM) can implement noise-handling by using a measure of information compression, an approach recently elaborated in Muggleton et al. [1992b] and Srinivasan et al. [1992].

One of the main concerns in our work on ILP has been the development and use of noise-handling techniques in empirical ILP. These issues are addressed in detail in Part II of the book.

1.6 Applications of ILP

An important application of ILP is knowledge acquisition in second generation expert systems, which use a first-order representation and possibly a deep model of the domain. A novel type of application is knowledge discovery in databases [Piatetsky and Frawley 1991, Frawley et al. 1991]. Finally, ILP can be used as a tool in various steps of scientific discovery, e.g., theory formation, design of experiments and theory verification [De Raedt 1993].

This section briefly outlines the role of ILP in knowledge acquisition, knowledge discovery in databases and scientific discovery. Other applications are only briefly mentioned here. An overview of several state-of-the-art ILP applications is given in Part III of the book.

Knowledge acquisition

The main problem to be faced when building expert systems is the acquisition of knowledge. Knowledge acquisition takes a lot of time since it is necessary to observe and interview domain experts who are often unable to formulate their expertise in a form suitable for machine use. This problem is known as the *knowledge acquisition bottleneck*. Machine learning contributes to the widening of this bottleneck by the development of tools that assist and partially automate the knowledge acquisition process. State-of-the-art inductive learning technology can be used to construct expert knowledge bases more effectively than traditional dialogue-based techniques for knowledge acquisition [Bratko 1992]. Currently, two trends have become prominent in this respect.

On the one hand, inductive logic programming is concerned with the use of more powerful concept description languages, facilitating the use of domain-specific knowledge – this is the main issue addressed in this book. ILP can be used to automatically construct knowledge bases for expert systems based on deep and qualitative models, for which propositional learning techniques are not sufficient.

An effective ILP knowledge acquisition tool is the system MOBAL [Morik 1991, Kietz and Wrobel 1992]. MOBAL includes a model acquisition tool which abstracts rule models from rules, a sort taxonomy tool which clusters constant terms occurring as arguments in training examples, and a predicate structuring tool which abstracts rule sets to a topology hierarchy. Another learning apprentice system that can be used as an interactive aid for building and refining a knowledge base is DISCIPLE [Tecuci and Kodratoff 1990] which has three integrated learning modules, a knowledge base consisting of action models and object knowledge and an expert system shell.

On the other hand, significant effort has been devoted to the development of toolkits of inductive learning techniques [Morik et al. 1991]. It has been recognized previously in the expert systems community that it is hard to solve the knowledge acquisition problem by using one individual technique and that it is more appropriate to have a 'toolkit' of techniques and select one or more of them according to the nature of the domain and type of knowledge under investigation [Harmon et al. 1988]. The machine learning community has come to the same conclusions, which led to the Machine Learning Toolbox (MLT) ESPRIT Project P2154 [Kodratoff et al. 1992] aimed at developing a workbench of machine learning tools, from which one or more can be selected in order to find the best solution to a specific problem. The *multistrategy learning* paradigm has also emerged recently [Michalski and Tecuci 1991]. A multistrategy learning system typically integrates two or more inference types (e.g., inductive, deductive, abductive) and/or computational mechanisms.

The LINUS environment [Lavrač et al. 1991a] presented in this book integrates various attribute-value inductive learners within a common ILP framework. It can thus be viewed as a toolkit of learning techniques, similar to MLT. The learning systems in the toolkit are a decision tree induction system and two rule induction systems. LINUS

contributes to attribute-value learners the ILP framework which allows for learning relations in the presence of background knowledge. To ILP, LINUS brings the techniques for handling imperfect (noisy) data and the potential for practical applications.

Finally, let us mention that the empirical ILP approach to knowledge acquisition is more appropriate for domains where relatively well established domain (background) knowledge is available in addition to a large number of examples of a single concept. For domains where multiple concepts have to be acquired and little (possibly incorrect) background knowledge is available, the interactive ILP paradigm seems to be more appropriate.

Knowledge discovery in databases

Knowledge discovery in databases is concerned with the non-trivial extraction of implicit, previously unknown, and potentially useful information from data stored in databases [Frawley et al. 1991, Piatetsky and Frawley 1991]. Frawley et al. [1991] indicate that one of the main problems in knowledge discovery in databases is the problem of noisy data. As the majority of practically applicable learning systems with noise-handling capabilities are attribute-value learners, research in knowledge discovery in databases has mainly taken place in the context of a single relation, in an attribute-value framework. The problem of inducing intensional relation definitions taking into account the dependencies on other relations has received attention only lately, as one of the main issues addressed in current ILP research.

Current empirical ILP systems already show the potential for knowledge discovery from large collections of data. Empirical ILP systems, such as FOIL, GOLEM and LINUS, are already efficient enough to be applied to real-life domains [Bratko 1992]. However, in order to discover interdependencies among different relations, multiple predicate learners should be preferably applied to the knowledge discovery task. Examples of multiple predicate learners that show the potential for knowledge discovery are MOBAL, MPL [De Raedt et al. 1993a] and CLAUDIEN [De Raedt and Bruynooghe 1993].

Inductive data engineering, an novel application of ILP for knowledge discovery, focuses on the inductive discovery of integrity constraints (similar as CLAUDIEN) as well as on data/knowledge base maintenance/design in accordance with these constraints [Flach 1993].

Scientific knowledge discovery

There exists a parallel between the discovery of scientific theories and the construction of knowledge bases for expert systems [De Raedt 1993]. The discovery process by a scientist and the knowledge acquisition process by a knowledge engineer usually rely on a number of observations or examples which they generalize using knowledge about the domain. The result is a theory which is initially unknown and can be considered a new piece or a new formulation of knowledge. In the generation of a new theory it is useful to perform experiments which would confirm or falsify the currently built theory. Also, the generated theory is in fact a hypothesis which has to be thoroughly tested. ILP can help the scientist and the knowledge engineer in all the indicated phases of theory construction.

ILP can be used in the following stages of scientific discovery:

- in the empirical or interactive generation of general logical theories from specific observations,
- in the interactive generation of crucial experiments while discovering a logical theory, and
- in the systematic testing of a given logical theory by proposing crucial experiments that would falsify the theory [De Raedt 1993].

The generation of logical theories from sets of observations can be performed by both empirical and interactive ILP systems. The empirical ILP system GOLEM already generated theories which are meaningful to scientists and have been published in scientific literature [Muggleton et al. 1992a, King et al. 1992]. Other systems that already show the potential of ILP for scientific discovery from data which may contain noise are FOIL (and mFOIL) and LINUS. The aspect of logical theory formation using empirical ILP systems is elaborated throughout this book.

The second issue, the design of experiments during scientific discovery, is addressed by interactive ILP systems. In this task it is important that all experiments performed are relevant and that they address

different settings explained by the theory. Examples of ILP systems potentially applicable for this task are MIS and CLINT. The third issue, the systematic testing of a hypothesized theory that falsifies it or partly confirms it, can also be addressed by interactive ILP systems [De Raedt et al. 1991].

Other applications

Among other ILP applications we first have to mention the software engineering task of inductive *logic program synthesis*. In this application, the task of an ILP learner is to induce a logic program from examples of program behavior. An initial, possibly incorrect program, may also be given. MIS [Shapiro 1983], one of the earliest ILP systems, was primarily applied to this problem (algorithmic program debugging). Another software engineering application of ILP is described in Bratko and Grobelnik [1993].

Another type of application, called *inductive engineering*, is concerned with the construction of a model/design of a device from a specification (examples) of its behavior [Mozetič 1987, Bratko et al. 1989]. This task is also known as design from first principles.

Both empirical and interactive ILP systems can be used in these application tasks. However, empirical ILP systems are more likely to be applied in practice for two reasons. First, there is more experience with learning single concepts from large collections of data than with deriving knowledge bases from a small number of examples. Second, empirical ILP systems are more efficient because of the use of heuristics, because there is no need to take into account dependencies among different concepts, and because no examples are generated [De Raedt and Bruynooghe 1992a].

Several applications illustrate the potential of ILP in learning from real-life data. These applications include learning qualitative models from sample behaviors [Bratko et al. 1991, Džeroski and Bratko 1992a], inducing temporal rules for satellite fault diagnosis [Feng 1991], predicting secondary structure of proteins [Muggleton et al. 1992a] and finite element mesh design [Dolšak and Muggleton 1992, Dolšak 1991, Džeroski and Dolšak 1991]. Some of these applications are outlined in Part III of the book.

2

Inductive logic programming

Inductive logic programming has its theoretical roots in computational logic [Lloyd 1990]. This chapter establishes the basic terminology used in logic programming and deductive databases and formally defines the tasks of empirical and interactive inductive logic programming. It also introduces a way of structuring the search space of program clauses based on the so-called θ-subsumption ordering of clauses introduced by Plotkin [1969].

2.1 Logic programming and deductive database terminology

This section briefly introduces the basic logic programming and deductive database terminology, which will be used throughout the book. For a comprehensive introduction to logic programming and the programming language Prolog, we refer the reader to Bratko [1990]. An introduction to deductive databases can be found in Ullman [1988]. A detailed theoretical treatment of logic programming and deductive databases can be found in Lloyd [1987]. The following definitions are abridged from Lloyd [1987], and take into account the Prolog syntax.

A *first-order alphabet* consists of variables, predicate symbols and function symbols (which include constants). A *variable* is represented by an upper case letter followed by a string of lower case letters and/or digits. A *function symbol* is a lower case letter followed by a string of lower case letters and/or digits. A *predicate symbol* is a lower case letter followed by a string of lower case letters and/or digits.

A variable is a *term*, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g and h are function symbols and X is a variable. A constant is a function symbol of arity 0, i.e., followed by a bracketed 0-tuple of terms. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an *atomic formula*, or *atom*. Both L and its *negation* $\overline{L}$ are *literals* whenever L is an atomic formula. In this case L is called a *positive literal* and $\overline{L}$ is called a *negative literal*.

A *clause* is a formula of the form

$$\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 \vee ... L_m)$$

where each L_i is a literal and $X_1, X_2, ..., X_s$ are all the variables occurring in $L_1 \vee L_2 \vee ... L_m$. A clause can also be represented as a finite set (possibly empty) of literals. Thus the set $\{L_1, L_2, .. \overline{L_i}, \overline{L_{i+1}}, ...\}$ stands for the clause $(L_1 \vee L_2 \vee .. \overline{L_i} \vee \overline{L_{i+1}} \vee ...)$ which is equivalently represented as $L_1 \vee L_2 \vee .. \leftarrow L_i \wedge L_{i+1} \wedge ...$. Most commonly, this same clause is written as $L_1, L_2, .. \leftarrow L_i, L_{i+1}, ...$, where commas on the left-hand side of $\leftarrow$ denote disjunctions, and commas on the right-hand side denote conjunctions. A set of clauses is called a *clausal theory* and represents the conjunction of its clauses.

Literals, clauses and clausal theories are all *well-formed formulae* (*wff*). Let E be a wff or term. Let $vars(E)$ denote the set of variables in E. E is said to be *ground* if and only if $\mathrm{vars}(E) = \emptyset$.

Horn clause A *Horn clause* is a clause which contains at most one positive literal.

Definite program clause A *definite program clause* is a clause which contains exactly one positive literal. It has the form

$$T \leftarrow L_1, \ldots, L_m$$

where $T, L_1, \ldots, L_m$ are atoms.

The positive literal (T) in a definite program clause is called the *head* of the clause. The conjunction of negative literals $(L_1, \ldots, L_m)$ is called the *body* of the clause. A Horn clause with no positive literal is a *definite goal*. A *positive unit clause* is a clause of the form $L \leftarrow$, that is, a definite program clause with an empty body. In the Prolog terminology, such a clause is called a *fact* and is denoted simply by L.

Definite program A set of definite program clauses is called a *definite logic program*.

Only atoms are allowed in the body of definite program clauses. In the programming language Prolog [Bratko 1990], however, literals of the form *not* L, where L is an atom, are allowed, where *not* is interpreted under the *negation-as-failure* rule [Clark 1978].

Program clause A *program clause* is a clause of the form

$$T \leftarrow L_1, \ldots, L_m \tag{2.1}$$

where T is an atom, and each of $L_1, \ldots, L_m$ is of the form L or *not* L, where L is an atom.

Normal program A *normal program* is a set of program clauses.

Predicate definition A *predicate definition* is a set of program clauses with the same predicate symbol (and arity) in their heads.

Let us now illustrate the above definitions on some simple examples. The clause

$$daughter(X, Y) \leftarrow female(X), mother(Y, X)$$

is a definite program clause, while the clause

$$daughter(X, Y) \leftarrow not\ male(X), father(Y, X)$$

is a (normal) program clause. Together, the two clauses constitute a predicate definition of predicate $daughter/2$. This predicate definition is also a normal program. The first clause is an abbreviated representation of the formula

$$\forall X \forall Y : daughter(X, Y) \vee \overline{female(X)} \vee \overline{mother(Y, X)}$$

and can also be written in set notation as

$$\{daughter(X, Y), \overline{female(X)}, \overline{mother(Y, X)}\}$$

Below we define three restricted forms of program clauses:

Datalog clause A *Datalog clause* is a program clause with no function symbols of non-zero arity. This means that only variables and constants can be used as predicate arguments.

Constrained clause A *constrained* clause is a program clause in which all variables in the body also appear in the head.

Non-recursive clause A *non-recursive* program clause is a program clause in which the predicate symbol in the head does not appear in any of the literals in the body.

Among the important developments in relational databases are *deductive databases* [Lloyd 1987, Ullman 1988] which allow for both extensional and intensional definitions of relations. The logic programming school in deductive databases [Lloyd 1987] argues that deductive databases can be effectively represented and implemented using logic and logic programming. The definitions below are adapted from Ullman [1988] and Lloyd [1987].

Relation A n-ary *relation* p is a set of tuples, i.e., a subset of the Cartesian product of n domains $D_1 \times D_2 \times \ldots \times D_n$, where a *domain* (or a *type*) is a set of values. It is assumed that a relation is finite unless stated otherwise.

Relational database A *relational database* (RDB) is a set of relations.

A relation in a database is essentially the same as a predicate in a logic program. The basic difference between program clauses (2.1) and database clauses (2.2), defined below, is in the use of types. A clause is *typed* if each variable appearing in arguments of clause literals is associated with a set of values the variable can take. For example, in the relation $lives\_in(X, Y)$, we may want to specify that X is of type *person* and Y is of type *city*.

It should be noted that types provide a simple syntactic way of specifying semantic information and can increase the deductive efficiency by eliminating useless branches of the search space. In logic, the terms *sort* and *many-sorted logic* are used, whereas in logic programming the term *type* is used instead of *sort* [Lloyd 1987]; we will adopt the term *type*.

Database clause A database clause is a typed *program clause* of the form:

$$T \leftarrow L_1, \ldots, L_m \tag{2.2}$$

where T is an atom and $L_1, \ldots, L_m$ are literals.

Deductive database – DDB A deductive database is a set of *database clauses*.

Database clauses use variables and function symbols in predicate arguments. As recursive types and recursive predicate definitions are allowed, the language is substantially more expressive than the language of relational databases. If we restrict database clauses to be non-recursive, we obtain the formalism of deductive hierarchical databases.

Deductive hierarchical database – DHDB A *deductive hierarchical database* is a deductive database restricted to non-recursive predicate definitions and non-recursive types.

Non-recursive types determine finite sets of values which are constants or structured terms with constant arguments. While the expressive power of DHDB is the same as that of RDB, DHDB allows intensional relations which can be much more compact than a RDB representation.

As stated earlier, a predicate definition is a set of program clauses. A predicate can be defined *extensionally* as a set of ground facts or *intensionally* as a set of database clauses. It is assumed that each predicate is defined either extensionally or intensionally. A relation in a database is essentially the same as a predicate definition in a logic program. Table 2.1 relates deductive database [Ullman 1988] and logic programming [Lloyd 1987] terms that will be used throughout the book.

DDB terminology	*LP terminology*
relation name p	*predicate symbol* p
attribute of relation p	*argument of predicate* p
tuple $< a_1, \ldots, a_n >$	*ground fact* $p(a_1, \ldots, a_n)$
relation p – *a set of tuples*	*definition of predicate* p – *a set of ground facts*

Table 2.1: Relating database and logic programming terms.

2.2 Empirical ILP

The definition of inductive concept learning with background knowledge, presented in the introductory chapter, reads as follows: Given a set of training examples $\mathcal{E}$ and background knowledge $\mathcal{B}$, find a hypothesis $\mathcal{H}$, expressed in some concept description language $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the background knowledge $\mathcal{B}$ and the examples $\mathcal{E}$. In inductive logic programming, $\mathcal{E}$ is a set of ground facts, where *positive examples* $\mathcal{E}^+$ are *true*, and *negative examples* $\mathcal{E}^-$ are *false* in the intended interpretation.

To make the above definition of learning operational for ILP, the following notion of *coverage* is used in the definitions of the function *covers* in equations (1.3) and (1.4).

Coverage Given background knowledge $\mathcal{B}$, hypothesis $\mathcal{H}$ and example set $\mathcal{E}$, hypothesis $\mathcal{H}$ covers example $e \in \mathcal{E}$ with respect to background knowledge $\mathcal{B}$ if $\mathcal{B} \cup \mathcal{H} \models e$, i.e.,

$$covers(\mathcal{B}, \mathcal{H}, e) = true \quad if \quad \mathcal{B} \cup \mathcal{H} \models e \tag{2.3}$$

Consequently, the function $covers(\mathcal{B}, \mathcal{H}, \mathcal{E})$ is defined as:

$$covers(\mathcal{B}, \mathcal{H}, \mathcal{E}) = \{e \in \mathcal{E} \mid \mathcal{B} \cup \mathcal{H} \models e\} \tag{2.4}$$

Again, completeness and consistency require that $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$, and $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-) = \emptyset$, respectively.

The semantic notion of coverage is used in the definitions of ILP in this chapter. It states that e is covered by $\mathcal{H}$ given background knowledge $\mathcal{B}$ if e is a logical consequence of $\mathcal{B} \cup \mathcal{H}$. This statement is denoted by $\mathcal{B} \cup \mathcal{H} \models e$, where the logical symbol $\models$ stands for semantic (logical) entailment [Lloyd 1987]. Note that, according to the definition of semantic entailment, $\mathcal{B} \cup \mathcal{H} \models e$ if $\mathcal{B} \cup \mathcal{H} \models_I e$ in every possible interpretation I, i.e., whenever an interpretation I makes $\mathcal{B} \cup \mathcal{H}$ true, it also makes e true.

In practice, for a given language bias $\mathcal{L}$, an appropriate proof procedure must be used to check whether an example is entailed by $\mathcal{B} \cup \mathcal{H}$. Most often, the SLD-resolution[1] proof procedure [Lloyd 1987] (with

---

[1] SLD-resolution stands for Linear resolution with Selection function for Definite clauses. SLD resolution is sound which means that for any wffs F and G it holds that $F \models G$ if G can be derived from F using SLD-resolution ($F \vdash_{SLD} G$).

bounded or unbounded depth) is used to check whether an example can be deduced from $\mathcal{B} \cup \mathcal{H}$. For instance, the systems MIS and CIGOL employ a depth bound in order to avoid infinite loops. In depth-bounded SLD-resolution, unresolved goals in the SLD-proof tree at depth h are not expanded and are treated as failed. Alternatively, a forward chaining procedure can be applied to check entailment [Bry 1990].[2]

The above notion of coverage is called *intensional coverage*, as the background knowledge $\mathcal{B}$ is intensional and can contain both ground facts and non-ground clauses. On the other hand, the *extensional notion of coverage* requires extensional background knowledge $\mathcal{B}$ in the form of ground facts only. Both notions were already introduced in MIS [Shapiro 1983], which can work extensionally (the *lazy* strategy) or intensionally (roughly the *adaptive* strategy). Similarly, different notions of coverage can be used in MARKUS [Grobelnik 1992]. Most empirical ILP systems use the extensional notion of coverage, while interactive ILP systems mostly use the intensional notion of coverage.

In the case that $\mathcal{B}$ consists of intensional predicate definitions, extensional ILP systems (such as FOIL [Quinlan 1990] and GOLEM [Muggleton and Feng 1990]) have to transform $\mathcal{B}$ into a ground (h-easy) model $\mathcal{M}$ of the background theory $\mathcal{B}$. This model contains all true ground facts that can be derived from $\mathcal{B}$ by a SLD-proof tree of depth less than h. Instead of $covers(\mathcal{B}, \mathcal{H}, e)$, extensional ILP systems employ the function $covers_{ext}(\mathcal{M}, \mathcal{H}, e)$ [De Raedt et al. 1993a], defined below, to test whether an example e is extensionally covered by $\mathcal{H}$.

Extensional coverage A hypothesis $\mathcal{H}$ extensionally covers an example e with respect to a ground model $\mathcal{M}$ if there exists a clause $c \in \mathcal{H}$, $c = T \leftarrow Q$, and a substitution θ, such that $T\theta = e$ and $Q\theta = \{L_1, \ldots, L_m\}\theta \subseteq \mathcal{M}$. This is denoted by $covers_{ext}(\mathcal{M}, \mathcal{H}, e)$.

The task of empirical ILP, which is concerned with learning a single predicate from a given set of examples, can now be formulated as follows.

---

[2] For a more detailed discussion of the proof procedures for different ILP systems see De Raedt [1992], pp. 75–88.

Empirical ILP

Given:

- a set of training examples $\mathcal{E}$, consisting of true $\mathcal{E}^+$ and false $\mathcal{E}^-$ ground facts of an unknown predicate p,
- a description language $\mathcal{L}$, specifying syntactic restrictions on the definition of predicate p, and
- background knowledge $\mathcal{B}$, defining predicates q_i (other than p) which may be used in the definition of p and which provide additional information about the arguments of the examples of predicate p.

Find:

- a definition $\mathcal{H}$ for p, expressed in $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the examples $\mathcal{E}$ and background knowledge $\mathcal{B}$.

In the following, we refer to the definition of p as the definition of the *target* relation. When learning from noisy examples, the completeness and consistency criteria need to be relaxed in order to avoid overly specific hypotheses [Lavrač and Džeroski 1992b].

Empirical ILP systems include FOIL [Quinlan 1990], mFOIL [Džeroski 1991], GOLEM [Muggleton and Feng 1990], ML-SMART [Bergadano and Giordana 1988], LINUS [Lavrač et al. 1991a], MARKUS [Grobelnik 1992] and RX [Tangkitvanich and Shimura 1992]. LINUS and FOIL upgrade attribute-value learners from the ID3 [Quinlan 1986] and AQ [Michalski 1969, Michalski et al. 1986] families towards a first-order logical framework. A different approach is used in GOLEM, which is based on Plotkin's notion of relative least general generalization (*rlgg*) [Plotkin 1969]. Another ILP system FOCL [Pazzani and Kibler 1992, Brunk and Pazzani 1991] is an extension of FOIL that combines empirical ILP and explanation based learning. The system MOBAL [Morik 1991, Kietz and Wrobel 1992] is an empirical system which learns multiple predicate definitions.

The complexity of learning grows with the expressiveness of the hypothesis language $\mathcal{L}$. Therefore, to reduce the complexity, various

restrictions can be applied to clauses expressed in $\mathcal{L}$. For example, some of the current empirical ILP systems constrain $\mathcal{L}$ to function-free program clauses, as is the case in FOIL. LINUS [Lavrač et al. 1991a] induces an even more restricted form of program clauses, i.e., deductive hierarchical database clauses.

The ij-determinacy bias in GOLEM [Muggleton and Feng 1990] is in fact a semantic bias as it depends not only on the form of hypotheses, but also on the background knowledge. Various other restrictions can be used including argument types and symmetry of predicates in pairs of arguments [Lavrač et al. 1991a], input/output modes [Shapiro 1983], program schemata or rule models [Wrobel 1988, Morik 1991], predicate sets [Bergadano et al. 1989], parametrized languages [De Raedt 1992], integrity constraints [De Raedt et al. 1991] and determinations [Russell 1989].

2.3 Interactive ILP

The problem of interactive ILP [De Raedt 1992], which is typically incremental, can be formalized as follows.

Interactive ILP

Given:

- a set of training examples $\mathcal{E}$ possibly about multiple predicates,
- background knowledge $\mathcal{B}$ consisting of predicate definitions assumed to be correct,
- a current hypothesis $\mathcal{H}$ which is possibly incorrect,
- a description language $\mathcal{L}$, specifying syntactic restrictions on the definitions of predicates in $\mathcal{H}$,
- an example e labeled $\oplus$ or $\ominus$, and
- an oracle willing to answer membership queries (label examples as $\oplus$ and $\ominus$) and possibly other questions.

Find:

- a new hypothesis $\mathcal{H}'$, obtained by retracting and asserting clauses belonging to $\mathcal{L}$ from $\mathcal{H}$ such that $\mathcal{H}'$ is complete and consistent with respect to the examples $\mathcal{E} \cup \{e\}$ and background knowledge $\mathcal{B}$.

Whereas empirical ILP is concerned with learning a single target relation from a large collection of possibly noisy examples, interactive ILP systems attempt to learn multiple relations, which may be interdependent, from a small set of correct examples. During the process of learning, interactive ILP systems generate their own examples and ask the user (oracle) about their label (classification $\oplus$ or $\ominus$). They may also ask the user about the validity of the generalizations constructed.

An important issue addressed in interactive ILP is the issue of representation change, i.e., shift of bias. Namely, several interactive ILP systems can change the hypothesis language $\mathcal{L}$ during the learning process. Representation changes include inventing new predicates as well as moving to a more powerful hypothesis language. This invokes the question of bias management, i.e., deciding whether, when and how to shift bias. The question of representation change has only recently been addressed in empirical ILP [Kijsirikul et al. 1992, Muggleton 1992b, Srinivasan et al. 1992].

An early interactive ILP system, MIS [Shapiro 1983], introduced the notion of refinement operator (graph) which is used to structure the space of clauses. MIS searches the refinement graph in a breadth-first top-down (general to specific) manner.

An important representative of interactive ILP systems is CLINT [De Raedt and Bruynooghe 1989, 1992a]. It is based on a sound theoretical framework and includes most of the above-mentioned characteristics of interactive ILP systems. CLINT generates its own examples and asks questions about their truth values, uses integrity constraints, manages a series of hypothesis languages and shifts among them as appropriate [De Raedt 1992].

Several interactive ILP systems are based on inverting the resolution principle [Robinson 1965]. Most of the work done in this area is a special form of the general theory of inverse resolution introduced in CIGOL [Muggleton and Buntine 1988] and refined in Muggleton [1991].

CIGOL employs three kinds of operators: absorption, which generalizes clauses; intraconstruction, which introduces new auxiliary predicates; and truncation, which generalizes unit clauses. The system MARVIN [Sammut and Banerji 1986] employs a variant of the absorption operator, the operators $G1$ and $G2$ introduced by Wirth [1989] are similar to absorption and intraconstruction, while IRES [Rouveirol 1990] and ITOU [Rouveirol 1992] overcome some of the problems with truncation and absorption by introducing a saturation procedure.

2.4 Structuring the hypothesis space

Concept learning can be viewed as a search problem [Mitchell 1982]. States in the search space (also called the *hypothesis space*) are concept descriptions and the goal is to find one or more states satisfying some quality criterion. A learner can be described in terms of the structure of its search space, its search strategy and search heuristics.

This section introduces the θ-subsumption lattice which gives the structure of the search space of program clauses. Having structured the hypothesis space, a learner can search it blindly, in an uninformed way, or heuristically. In uninformed search, the depth-first or breadth-first search strategy can be applied. On the other hand, in heuristic search, the usual search strategies are hill-climbing or beam search. A detailed analysis of heuristics used to guide the search of clauses can be found in Chapter 8.

In ILP, the search space is determined by the language of logic programs $\mathcal{L}$ consisting of program clauses of the form $T \leftarrow Q$, where T is an atom $p(X_1, \ldots, X_n)$ and Q is a conjunction of literals $L_1, \ldots, L_m$. The vocabulary of predicate symbols q_i in the literals L_i of the body of a clause is determined by the predicates from the background knowledge $\mathcal{B}$. If $\mathcal{L}$ allows for recursive predicate definitions, the predicate symbol in L_i can also be the predicate symbol p itself.

The selected language bias $\mathcal{L}$ syntactically restricts the form of clauses which can be formulated from a given vocabulary of predicates, function symbols and constants of the language. To constrain the search space, the language bias $\mathcal{L}$ in actual ILP systems is some restricted form of logic programs, e.g., consisting of function-free program clauses (in FOIL), constrained deductive hierarchical database clauses (in LINUS), or definite clauses (in GOLEM).

In order to search the space of program clauses systematically, it is useful to structure the search space by introducing a partial ordering into a set of clauses based on the θ-subsumption. To define θ-subsumption, we first need to define substitution:

Substitution A *substitution* $\theta = \{X_1/t_1, \ldots, X_k/t_k\}$ is a function from variables to terms. The application $W\theta$ of a substitution θ to a wff W is obtained by replacing all occurrences of each variable X_j in W by the same term t_j [Lloyd 1987].

θ-subsumption Let c and c' be two program clauses. Clause c *θ-subsumes* c' if there exists a substitution θ, such that $c\theta \subseteq c'$ [Plotkin 1969]. Two clauses c and d are *θ-subsumption equivalent* if c θ-subsumes d and d θ-subsumes c. A clause is *reduced* if it is not θ-subsumption equivalent to any proper subset of itself.

The following examples are used to illustrate the above notions.

Example 2.1 (Substitution, subsumption)
Let c be the clause:

$$c = daughter(X, Y) \leftarrow parent(Y, X)$$

A substitution $\theta = \{X/mary, Y/ann\}$ applied to clause c is obtained by applying θ to each of its literals:

$$c\theta = daughter(mary, ann) \leftarrow parent(ann, mary)$$

As pointed out in Section 2.1, the clausal notation

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

stands for

$$\{daughter(X, Y), \overline{parent(Y, X)}\}$$

where all variables are assumed to be universally quantified and the commas denote disjunction. According to the definition, clause c θ-subsumes c' if there is a substitution θ that can be applied to c such that every literal in the resulting clause occurs in c'.

Clause c θ-subsumes the clause

$$c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

under the empty substitution $\theta = \emptyset$, since the set

$$\{daughter(X, Y), \overline{parent(Y, X)}\}$$

is a proper subset of $\{daughter(X, Y), \overline{female(X)}, \overline{parent(Y, X)}\}$.

Clause c θ-subsumes the clause

$$c' = daughter(X, X) \leftarrow female(X), parent(X, X)$$

under the substitution $\theta = \{Y/X\}$.

Clause c θ-subsumes the clause

$$\begin{array}{lcl} daughter(mary, ann) & \leftarrow & female(mary), \\ & & parent(ann, mary), \\ & & parent(ann, tom) \end{array}$$

under the substitution $\theta = \{X/mary, Y/ann\}$.

Clauses $c = daughter(X, Y) \leftarrow parent(Y, X), parent(W, V)$ and $d = daughter(X, Y) \leftarrow parent(Y, X)$ θ-subsume one another, i.e., are θ-subsumption equivalent. Clause d is reduced, while clause c is not. ■

θ-subsumption introduces the syntactic notion of generality. Clause c *is at least as general as* clause c' ($c \leq c'$) if c *θ-subsumes* c'. Clause c *is more general than* c' ($c < c'$) if $c \leq c'$ holds and $c' \leq c$ does not. In this case, we say that c' is a *specialization (refinement)* of c and c is a *generalization* of c'. The only clause refinements usually considered by the learner are the minimal (most general) specializations of the clause.

There are two important properties of θ-subsumption:

- If c θ-subsumes c' then c logically entails c', $c \models c'$. The reverse is not always true. As an example, Flach [1992] gives the following two clauses: $c = list([V|W]) \leftarrow list(W)$ and $c' = list([X, Y|Z]) \leftarrow list(Z)$. Given the empty list, c constructs lists of any given length, while c' constructs lists of even length only, and thus $c \models c'$. However, no substitution exists that can be applied to c to yield c', since it should map W both to $[Y|Z]$ and to Z which is impossible. Therefore, c does not θ-subsume c'.

- The relation $\leq$ introduces a *lattice* on the set of reduced clauses. This means that any two clauses have a least upper bound (lub) and a greatest lower bound (glb). Both the lub and glb are unique up to renaming of variables.

The second property of θ-subsumption leads to the following definition:

Least general generalization The *least general generalization* (lgg) of two reduced clauses c and c', denoted by $lgg(c, c')$, is the least upper bound of c and c' in the θ-subsumption lattice.

The rules for computing the lgg of two clauses are outlined in Section 3.1.1.

Note that θ-subsumption and least general generalization are purely syntactic notions since they do not take into account any background knowledge. Their computation is therefore simple and easy to implement in an ILP system. The same holds for the notion of generality based on θ-subsumption. On the other hand, taking background knowledge into account would lead to the notion of *semantic generality* [Niblett 1988, Buntine 1988]. Clause c is *at least as general* as clause c' with respect to background theory $\mathcal{B}$ if $\mathcal{B} \cup \{c\} \models c'$. The syntactic, θ-subsumption based, generality is computationally more feasible. Namely, semantic generality is in general undecidable and does not introduce a lattice on a set of clauses. Because of these problems, syntactic generality is more frequently used in ILP systems.

θ-subsumption is important for learning for the following reasons:

- As shown above, it provides a generality ordering for hypotheses, thus structuring the hypothesis space.
- It can be used to prune large parts of the search space.
    - When generalizing c to c', $c' < c$, all the examples covered by c will also be covered by c' (since if $\mathcal{B} \cup \{c\} \models e$ holds then also $\mathcal{B} \cup \{c'\} \models e$ holds). This property is used to prune the search of more general clauses when e is a negative example: if c is inconsistent (covers a negative example) then all its generalizations will also be inconsistent. Hence, the generalizations of c do not need to be considered.
    - When specializing c to c', $c < c'$, an example not covered by c will not be covered by any of its specializations either (since if $\mathcal{B} \cup \{c\} \not\models e$ holds then also $\mathcal{B} \cup \{c'\} \not\models e$ holds). This property is used to prune the search of more specific clauses when e is an uncovered positive example: if c does not cover a positive example none of its specializations will. Hence, the specializations of c do not need to be considered.

θ-subsumption provides the basis for two important ILP techniques:

- bottom-up *building of least general generalizations* from training examples, relative to background knowledge, and
- top-down *searching of refinement graphs*.

These two techniques and inverse resolution will be elaborated in the following chapter.

3

Basic ILP techniques

A search of the hypothesis space can be performed bottom-up or top-down. Generalization techniques search the hypothesis space in a bottom-up manner: they start from the training examples (most specific hypotheses) and search the hypothesis space by using generalization operators. Generalization techniques are best suited for interactive and incremental learning from few examples. On the other hand, specialization techniques search the hypothesis space top-down, from the most general to specific concept descriptions, using specialization operators. Specialization techniques are better suited for empirical learning in the presence of noise since top-down search can easily be guided by heuristics. This chapter introduces the basic generalization and specialization ILP techniques. An overview of the basic ILP techniques is also given in Flach [1992]. We further describe the unifying framework for generalization introduced by Muggleton [1991], as well as a unifying framework for specialization [Lavrač et al. 1992b] which is related to the GENCOL framework [De Raedt and Bruynooghe 1992b].

3.1 Generalization techniques

Generalization techniques search the hypothesis space in a bottom-up manner. Bottom-up learners start from the most specific clause (allowed by the language bias) that covers a given example and then generalize the clause until it cannot further be generalized without covering negative examples.

In bottom-up hypothesis generation, a generalization c' of clause c ($c' < c$) can be obtained by applying a θ-subsumption-based *generalization operator*.

Generalization operator Given a language bias $\mathcal{L}$, a generalization operator ρ maps a clause c to a set of clauses $\rho(c)$ which are generalizations of c:

$$\rho(c) = \{c' \mid c' \in \mathcal{L},\ c' < c\}$$

Generalization operators perform two basic syntactic operations on a clause:

1. apply an inverse substitution to the clause, and
2. remove a literal from the body of the clause.

In Sections 3.1.1 and 3.1.2 we briefly sketch two basic generalization techniques: *relative least general generalization* (*rlgg*, used in GOLEM) and *inverse resolution* (used in CIGOL), which are both used in hypothesis generation.

Muggleton [1991] introduced a unifying framework covering both relative least general generalization and inverse resolution, based on the notion of a *most specific inverse resolvent*. This framework is briefly introduced in Section 3.1.3. In this unifying framework, an *rlgg* is an *lgg* of most specific inverse linear derivations. This is a first step towards a unifying theory of empirical and interactive ILP. Another step towards integrating empirical and interactive ILP is the recent work on non-interactive theory revision [Mooney and Richards 1992, Tangkitvanich and Shimura 1992] and multiple predicate learning [De Raedt et al. 1993a].

3.1.1 Relative least general generalization

Plotkin's notion of least general generalization (*lgg*) [Plotkin 1969] is important for ILP since it forms the basis of cautious generalization algorithms which perform bottom-up search of the θ-subsumption lattice. Cautious generalization assumes that if two clauses c_1 and c_2 are true, it is very likely that $lgg(c_1, c_2)$ will also be true.

Let us recall the definition of the least general generalization of two clauses c and c' from Section 2.4: $lgg(c, c')$ is the least upper bound of c and c' in the θ-subsumption lattice. To actually compute the *lgg* of two clauses, *lgg* of terms, atoms and literals need to be defined first [Plotkin 1969].

lgg of terms $lgg(t_1, t_2)$

1. $lgg(t, t) = t$,
2. $lgg(f(s_1, ..., s_n), f(t_1, ..., t_n)) = f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$,
3. $lgg(f(s_1, ..., s_m), g(t_1, ..., t_n)) = V$, where $f \neq g$, and V is a variable which represents $lgg(f(s_1, ..., s_m), g(t_1, ..., t_n))$,
4. $lgg(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable; in this case, V is a variable which represents $lgg(s, t)$.

Examples:

$lgg([a, b, c], [a, c, d]) = [a, X, Y]$.
$lgg(f(a, a), f(b, b)) = f(lgg(a, b), lgg(a, b)) = f(V, V)$ where V stands for $lgg(a, b)$. When computing *lgg*s one must be careful to use the same variable for multiple occurrences of the *lgg*s of subterms, i.e., $lgg(a, b)$ in this example. This holds for *lgg*s of terms, atoms and clauses alike.

lgg of atoms $lgg(A_1, A_2)$

1. $lgg(p(s_1, ..., s_n), p(t_1, ..., t_n)) = p(lgg(s_1, t_1), ..., lgg(s_n, t_n))$, if atoms have the same predicate symbol p,
2. $lgg(p(s_1, ..., s_m), q(t_1, ..., t_n))$ is undefined if $p \neq q$.

lgg of literals $lgg(L_1, L_2)$

1. if L_1 and L_2 are atoms, then $lgg(L_1, L_2)$ is computed as defined above,
2. if both L_1 and L_2 are negative literals, $L_1 = \overline{A_1}$ and $L_2 = \overline{A_2}$, then $lgg(L_1, L_2) = lgg(\overline{A_1}, \overline{A_2}) = \overline{lgg(A_1, A_2)}$,
3. if L_1 is a positive and L_2 is a negative literal, or vice versa, $lgg(L_1, L_2)$ is undefined.

Examples:

$lgg(parent(ann, mary), parent(ann, tom)) = parent(ann, X)$.
$lgg(parent(ann, mary), \overline{parent(ann, tom)}) = undefined$.
$lgg(parent(ann, X), daughter(mary, ann)) = undefined$.

lgg of clauses $lgg(c_1, c_2)$

Let $c_1 = \{L_1, ..., L_n\}$ and $c_2 = \{K_1, ..., K_m\}$. Then $lgg(c_1, c_2) = \{L_{ij} = lgg(L_i, K_j) \mid L_i \in c_1, K_j \in c_2$ and $lgg(L_i, K_j)$ is defined $\}$.

Examples:

If $c_1 = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$ and $c_2 = daughter(eve, tom) \leftarrow female(eve), parent(tom, eve)$, then $lgg(c_1, c_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$, where X stands for $lgg(mary, eve)$ and Y stands for $lgg(ann, tom)$.

The definition of *relative least general generalization* can now be introduced.

Relative least general generalization The *relative least general generalization* (*rlgg*) of two clauses c_1 and c_2 is their least general generalization $lgg(c_1, c_2)$ *relative* to background knowledge $\mathcal{B}$.

This technique will be discussed in some more detail in Section 4.2, which gives an overview of GOLEM. In short, if the background knowledge consists of ground facts, and K denotes the conjunction of all these facts, the *rlgg* of two ground atoms A_1 and A_2 (positive examples), relative to the given background knowledge K, is defined as:

$$rlgg(A_1, A_2) = lgg((A_1 \leftarrow K), (A_2 \leftarrow K))$$

Example 3.1 (Relative least general generalization)

Given the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ and the background knowledge $\mathcal{B}$ for the family example in Section 1.4, the least general generalization of e_1 and e_2 relative to $\mathcal{B}$ is computed as:

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

where K denotes the conjunction of the literals $parent(ann, mary)$, $parent(ann, tom)$, $parent(tom, eve)$, $parent(tom, ian)$, $female(ann)$, $female(mary)$, and $female(eve)$.

The result of computing the $rlgg(e_1, e_2)$ can be found in Section 4.2. Finally, after the elimination of irrelevant literals, the following clause is generated by GOLEM:

$$rlgg(e_1, e_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

■

3.1.2 Inverse resolution

The basic idea of *inverse resolution*, introduced as a generalization technique to ILP by Muggleton and Buntine [1988], is to invert the *resolution* rule of deductive inference [Robinson 1965], e.g., to invert the SLD-resolution proof procedure for definite programs [Lloyd 1987]. The basic resolution step in propositional logic derives the resolvent $p \vee r$ given the premises $p \vee \overline{q}$ and $q \vee r$.

Example 3.2 (A propositional derivation tree)
Given the theory $\mathcal{T} = \{u \leftarrow v, v \leftarrow w, w\}$ suppose we want to derive u. Proposition w resolves with $v \leftarrow w$ to give v, which is then resolved with $u \leftarrow v$ to derive u. The derivation tree for this simple example is given in Figure 3.1.

■

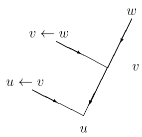

Figure 3.1: A simple propositional derivation tree.

The above example illustrates resolution in the propositional case. In a first-order case, resolution is more complicated, involving substitutions. Let $res(c, d)$ denote the resolvent of clauses c and d.

Example 3.3 (A first-order derivation tree)
To illustrate resolution in first-order logic, we again use the family example from Section 1.4. Suppose that background knowledge $\mathcal{B}$ consists of the clauses $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$ and $\mathcal{H} = \{c\} = \{daughter(X,Y) \leftarrow female(X), parent(Y,X)\}$. Let $\mathcal{T} = \mathcal{H} \cup \mathcal{B}$. Suppose we want to derive the fact $e = daughter(mary, ann)$ from $\mathcal{T}$. To this end, we proceed as follows:

- First, the resolvent $c_1 = res(c, b_1)$ is computed under the substitution $\theta_1 = \{X/mary\}$. This means that the substitution θ_1 is first applied to clause c to obtain $daughter(mary, Y) \leftarrow female(mary), parent(Y, mary)$, which is then resolved with b_1 as in the propositional case. The resolvent of $daughter(X,Y) \leftarrow female(X), parent(Y,X)$ and $female(mary)$ is thus $c_1 = res(c, b_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$.
- The next resolvent $c_2 = res(c_1, b_2)$ is computed under the substitution $\theta_2 = \{Y/ann\}$. The clauses $daughter(mary, Y) \leftarrow parent(Y, mary)$ and $parent(ann, mary)$ resolve in $e = res(c_1, b_2)$ $= daughter(mary, ann)$.

The linear derivation tree for this resolution process is given in Figure 3.2.

■

Inverse resolution, used in the interactive ILP system CIGOL [Muggleton and Buntine 1988], inverts the resolution process using the generalization operator based on inverting substitution [Buntine 1988]. Given a wff W, an *inverse substitution* θ^{-1} of a substitution θ is a function that maps terms in $W\theta$ to variables, such that $W\theta\theta^{-1} = W$.

Example 3.4 (Inverse substitution – a simple example)
Let $c = daughter(X,Y) \leftarrow female(X), parent(Y,X)$ and the substitution $\theta = \{X/mary, Y/ann\}$:

$$c' = c\theta = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$$

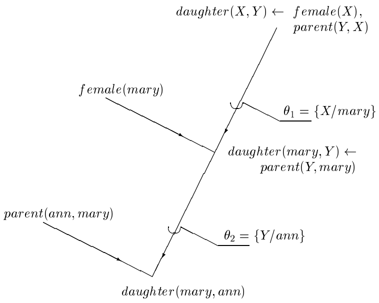

Figure 3.2: A first-order linear derivation tree.

By applying the inverse substitution $\theta^{-1} = \{mary/X, ann/Y\}$ the original clause c is obtained:

$$c = c'\theta^{-1} = daughter(X,Y) \leftarrow female(Y), parent(Y,X)$$

■

In the general case, inverse substitution is substantially more complex. It involves the *places* of terms in order to ensure that the variables in the initial wff W are appropriately restored in $W\theta\theta^{-1}$.

Example 3.5 (Inverse substitution – with places)
Let $W = loves(X, daughter(Y))$ and $\theta = \{X/ann, Y/ann\}$. Applying the substitution to W results in $W\theta = loves(ann, daughter(ann))$. The inverse substitution $\theta^{-1} = \{(ann, \{< 1 >\})/X, (ann, \{< 2, 1 >\})/Y\}$ specifies that the first occurrence (at *place* $< 1 >$) of the subterm ann in the term $W\theta$ is replaced by variable X and the second occurrence (at place $< 2, 1 >$) is replaced by Y. The use of places in θ^{-1} ensures that $W\theta\theta^{-1} = loves(X, daughter(Y)) = W$.

■

Inverse resolution will not be elaborated theoretically, but will rather be illustrated by an example. Let $ires(c, d)$ denote the inverse resolvent of clauses c and d.

Example 3.6 (Inverse resolution)
As in Example 3.3, let background knowledge $\mathcal{B}$ consist of the two clauses $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$. Let the current hypothesis $\mathcal{H} = \emptyset$. Suppose that the learner encounters the positive example $e_1 = daughter(mary, ann)$. The inverse resolution process might then proceed as follows:

- In the first step, inverse resolution attempts to find a clause c_1 which will, together with b_2, entail e_1 and can be added to the current hypothesis $\mathcal{H}$ instead of e_1. Choosing the inverse substitution $\theta_2^{-1} = \{ann/Y\}$, an inverse resolution step generates the clause $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$. Clause c_1 becomes the current hypothesis $\mathcal{H}$, such that $\{b_2\} \cup \mathcal{H} \models e_1$.

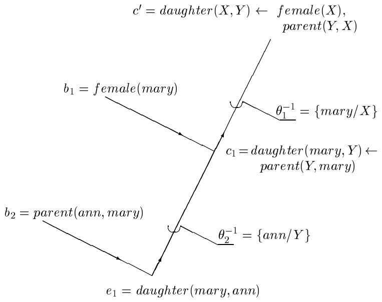

Figure 3.3: An inverse linear derivation tree.

- Inverse resolution then takes $b_1 = female(mary)$ and the current hypothesis $\mathcal{H} = \{c_1\} = \{daughter(mary, Y) \leftarrow parent(Y, mary)\}$. By computing $c' = ires(b_1, c_1)$, using the inverse substitution $\theta_1^{-1} = \{mary/X\}$, it generalizes the clause c_1 with respect to background knowledge $\mathcal{B}$, yielding $c' = daughter(X,Y) \leftarrow female(X), parent(Y,X)$. In the current hypothesis $\mathcal{H}$, clause c_1 can now be replaced by the more general clause c' which together with $\mathcal{B}$ entails the example e_1. Thus, the induced hypothesis is $\mathcal{H} = \{daughter(X,Y) \leftarrow female(X), parent(Y,X)\}$.

The corresponding inverse linear derivation tree is illustrated in Figure 3.3.

■

The generalization operator illustrated in the above example is called the *absorption* operator (the **V** operator), and is used in the ILP systems MARVIN [Sammut and Banerji 1986] and CIGOL [Muggleton and Buntine 1988] to generate hypotheses. Several other inverse resolution operators are used in CIGOL. An important operator is *intra-construction*, an instance of the **W** operator, which combines two **V** operators. This operator generates clauses using predicates which are not available in the initial learner's vocabulary. The ability to introduce new predicates in ILP is referred to as *predicate invention*.

3.1.3 A unifying framework for generalization

Inverse resolution is non-deterministic. Typically, at each inverse resolution step various generalizations of a clause can be performed, depending on the choice of the clause to be inversely resolved upon and the employed inverse substitution.

Example 3.7 (Non-determinism of inverse resolution)
To illustrate the problem of non-determinism, let us again take the setting of Example 3.6 where $e_1 = daughter(mary, ann)$ is the clause to be inversely resolved upon and $\mathcal{B}$ consists of clauses b_1 and b_2.

- First, at each inverse resolution step various clauses from $\mathcal{B}$ can be selected. Suppose that b_2 is selected first.

- Second, the question arises whether the generalization should be conservative, using an empty inverse substitution, or less cautious. In Example 3.6, given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$, the inverse substitution $\theta_2^{-1} = \{ann/Y\}$ was used to generalize e_1 to clause $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$. A more cautious generalization could be achieved by using the *most specific inverse substitution*, namely the empty inverse substitution $\theta_2^{-1} = \emptyset$ which would lead to the most specific inverse resolvent of b_2 and e_1: $c_1 = ires_{ms}(b_2, e_1) = daughter(mary, ann) \leftarrow parent(ann, mary)$.

■

To overcome the problem of non-determinism, Muggleton's unifying framework for relative least general generalization and inverse resolution [Muggleton 1991] proposes that *most specific inverse resolution* be used in the generalization process, using the *most specific inverse substitution* at each inverse resolution step. Let $ires_{ms}(c, d)$ denote the most specific inverse resolvent of clauses c and d.

Example 3.8 (Most specific inverse resolution)
Given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$, using the empty substitution $\theta_2^{-1} = \emptyset$, the most specific inverse resolvent $c_1 = ires_{ms}(b_2, e_1)$ is constructed:

$$c_1 = daughter(mary, ann) \leftarrow parent(ann, mary)$$

In the next inverse resolution step, the most specific inverse resolvent c' of c_1 and b_1 is generated, $c' = ires_{ms}(b_1, c_1)$, again using the empty substitution $\theta_1^{-1} = \emptyset$:

$$c' = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$$

The *most specific inverse linear derivation tree* for this example is given in Figure 3.4.

■

Muggleton's unifying framework relates two generalization techniques: inverse resolution and relative least general generalization. More specifically, relative least general generalizations are least general generalizations of most specific inverse linear derivations. The main idea of this framework is illustrated by an example.

Example 3.9 (A unifying framework for generalization)
To illustrate Muggleton's unifying framework, let us assume that $\mathcal{B}$ consists of four clauses: b_1, b_2 as in Examples 3.8 and 3.6, and two additional clauses $b_3 = female(eve)$ and $b_4 = parent(tom, eve)$. Suppose that two positive training examples are given $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$. Two most specific inverse linear derivation trees are generated.

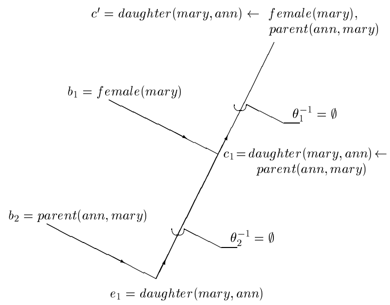

Figure 3.4: A most specific inverse linear derivation tree.

- The first tree is the same as in Example 3.8 illustrated in Figure 3.4. The most specific inverse resolution results in the clause:

  $$c' = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$$

- The second tree is generated analogously:
  - Under the empty inverse substitution, e_2 is first inversely resolved with b_4 into $c_3 = ires_{ms}(b_4, e_2)$:

    $$c_3 = daughter(eve, tom) \leftarrow parent(tom, eve)$$

  - Next, $c'' = ires_{ms}(b_3, c_3)$ is computed under the empty inverse substitution:

    $$c'' = daughter(eve, tom) \leftarrow female(eve), parent(tom, eve)$$

- Finally, the clause c is computed as the least general generalization of c' and c''. Clause $c = lgg(c', c'')$ generalizes e_1 and e_2.

  $$c = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

The above process is illustrated in Figure 3.5.

In Section 3.1.1 we have outlined the method for computing the relative least general generalization of two examples e_1 and e_2:

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

In Example 3.9, K is the conjunction of clauses b_1, b_2, b_3 and b_4. The result of computing the $rlgg(e_1, e_2)$ is, after the elimination of irrelevant literals (see Section 4.2), the same as the lgg of the most specific inverse linear derivations of e_1 and e_2:

$$c = rlgg(e_1, e_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

■

Example 3.9 illustrates only the basic idea of the unifying framework for bottom-up generalization methods as used in CIGOL and GOLEM. In summary, relative least general generalizations are least general generalizations of most specific inverse linear derivations. The reader can find more details in Muggleton [1991].

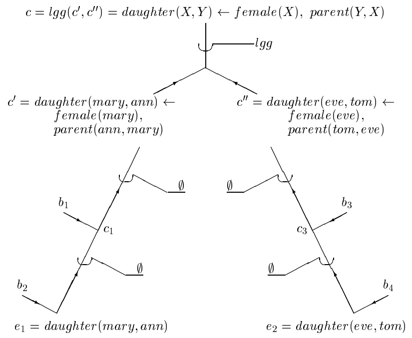

Figure 3.5: Muggleton's unifying framework for generalization.

3.2 Specialization techniques

Specialization techniques search the hypothesis space in a top-down manner, from general to specific hypotheses, using a θ-subsumption-based *specialization operator*. A specialization operator is usually called a *refinement operator*.

Refinement operator Given a language bias $\mathcal{L}$, a refinement operator ρ maps a clause c to a set of clauses $\rho(c)$ which are specializations (refinements) of c:

$$\rho(c) = \{c' \mid c' \in \mathcal{L},\ c < c'\}$$

A refinement operator typically computes only the set of minimal (most general) specializations of a clause under θ-subsumption. It employs two basic syntactic operations on a clause:

1. apply a substitution to the clause, and
2. add a literal to the body of the clause.

The basic specialization ILP technique is *top-down search of refinement graphs*. Top-down learners start from the most general clauses and repeatedly refine (specialize) them until they no longer cover negative examples; during the search they ensure that the clauses considered cover at least one positive example.

This ILP technique was first used in the interactive system MIS (Model Inference System [Shapiro 1983]). Section 3.2.1 briefly illustrates this technique on a simple example using an interactive MIS-like algorithm. On the other hand, top-down search of refinement graphs is the main technique used in empirical ILP. A unifying framework for empirical ILP algorithms which perform top-down search of refinement graphs is given in Section 3.2.2.

3.2.1 Top-down search of refinement graphs

For a selected language bias $\mathcal{L}$ and given background knowledge $\mathcal{B}$, the hypothesis space of program clauses is a lattice, structured by the θ-subsumption generality ordering. In this lattice, a *refinement graph* can be defined and used to direct the search from most general to specific hypotheses. A refinement graph is a directed, acyclic graph in which *nodes* are program clauses and *arcs* correspond to the basic refinement operations: substituting a variable with a term, and adding a literal to the body of a clause.

Algorithm 3.1 outlines the basic steps of the MIS algorithm [Shapiro 1983]. It assumes the language $\mathcal{L}$ of definite clauses, and the definitions of background knowledge predicates $\mathcal{B}$.

Algorithm 3.1 (A simplified MIS)

```
Initialize hypothesis $\mathcal{H}$ to a (possibly empty) set of clauses in $\mathcal{L}$.
repeat
  Read the next (positive or negative) example.
  repeat
    if there exists a covered negative example $e$ then
      Delete incorrect clauses from $\mathcal{H}$.
    if there exists a positive example $e$ not covered by $\mathcal{H}$ then
      With breadth-first search of the refinement graph
        develop a clause $c$ which covers $e$ and add it to $\mathcal{H}$.
  until $\mathcal{H}$ is complete and consistent.
  Output: Hypothesis $\mathcal{H}$.
forever
```

MIS interactively accepts new training examples. In the first **if** statement of the internal **repeat** loop, all the incorrect clauses are deleted from $\mathcal{H}$. Clause c in $\mathcal{H}$ is *incorrect* if c is responsible for $\mathcal{H}$ being inconsistent.

The main part of the algorithm is clause construction in the second **if** statement of the internal **repeat** loop: searching for a new clause of the target predicate definition that covers a positive example not covered by the current hypothesis. Search of the refinement graph starts with the most general clause and continues by searching clause refinements in a breadth-first manner. At each step, all minimal refinements are generated and tested for coverage. The acceptable refinements must cover the selected positive example. The process stops when the first acceptable consistent clause is found.

In Example 3.10 a MIS-like algorithm is illustrated. For simplicity, the language $\mathcal{L}$ is restricted to non-recursive definite clauses and only one specialization operation is used, namely, adding a literal to the body of a clause.

Example 3.10 (Searching the refinement graph)
Let us again take the family relations example from Section 1.4. The learner is first given the positive example $e_1 = daughter(mary, ann)$. Formally, the top-level node in the refinement graph is the most general clause $false$ (denoted by $\Box$, or alternatively, by the empty clause $\leftarrow$ which stands for $false \leftarrow true$). Practically, the search for clauses starts with the most general definition of the predicate $daughter$:

$$c = daughter(X,Y) \leftarrow$$

where an empty body is written instead of the body $true$. Since c covers e_1, the current hypothesis $\mathcal{H}$ is initialized to $\mathcal{H} = \{c\}$. The second example encountered $e_2 = daughter(eve, tom)$ is also positive and is covered by $\mathcal{H}$; therefore, the hypothesis does not need to be modified.

The third example encountered $e_3 = daughter(tom, ann)$ is negative. The learner deletes clause c from $\mathcal{H}$, since it covers the negative example. It now enters the second **if** statement in order to generate a new clause that covers the first positive example e_1. Since c covers the negative example e_3, the learner now generates the set of its refinements (minimal specializations) of the form

$$\rho(c) = \{daughter(X,Y) \leftarrow L\}$$

where L is one of following literals:[1]

- literals having as arguments the variables from the head of the clause: $X = Y$, $female(X)$, $female(Y)$, $parent(X,X)$, $parent(X,Y)$, $parent(Y,X)$, and $parent(Y,Y)$, and
- literals that introduce a new distinct variable Z ($Z \neq X$ and $Z \neq Y$) in the clause body: $parent(X,Z)$, $parent(Z,X)$, $parent(Y,Z)$, and $parent(Z,Y)$,

---

[1] Since the language is restricted to definite clauses, literals of the form *not* L are not considered. Also, as it is restricted to non-recursive clauses, literals with the target predicate $daughter/2$ are not considered.

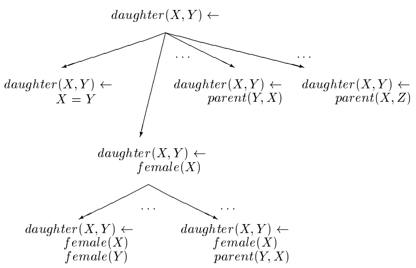

Figure 3.6: Part of the refinement graph for the family relations problem.

Part of the refinement graph for the family relations problem is given in Figure 3.6.

The refinements listed above are now considered one by one. The refinement $daughter(X,Y) \leftarrow X = Y$ is considered first, but is retracted as it does not cover the positive example e_1. The refinement $c' = daughter(X,Y) \leftarrow female(X)$ is considered next and is found to discriminate between positive and negative examples. As such, it is retained in the current hypothesis $\mathcal{H}$. This hypothesis would be the result of learning in the case that the learner encounters no more examples. However, suppose that another negative example $e_4 = daughter(eve, ann)$ is encountered. Clause c' then becomes inconsistent, since it covers e_4. It is deleted from $\mathcal{H}$ and the current hypothesis is again $\mathcal{H} = \emptyset$.

The clause generation process described above is now repeated, and the system tries again to build a clause that covers e_1. In this process, the remaining refinements of c are first considered. Of these, $daughter(X,Y) \leftarrow female(Y)$ and $daughter(X,Y) \leftarrow parent(Y,X)$ are retained in the breadth-first queue since they cover e_1, but are not consistent. The refinements of the clause $daughter(X,Y) \leftarrow X = Y$ are considered next, and are all retracted as they do not cover e_1. Finally, the refinements of c' are considered, among which $daughter(X,Y) \leftarrow female(X), parent(Y,X)$ covers e_1 and is consistent. As it happens, this clause also covers e_2, so the learner stops the search for clauses. ■

3.2.2 A unifying framework for specialization

Most top-down empirical ILP systems are based on the top-down search of refinement graphs, used in MIS. These include FOIL [Quinlan 1990], its successors mFOIL [Džeroski 1991] and FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], MARKUS [Grobelnik 1992], CLAUDIEN [De Raedt and Bruynooghe 1993] and MPL [De Raedt et al. 1993a].

This section presents a generic top-down empirical ILP algorithm which builds a unifying framework for most of the above empirical ILP systems. In order to present the algorithm the basic notation and terminology need to be established.

- A hypothesis $\mathcal{H} \in \mathcal{P}(\mathcal{L})^2$ is a set of program clauses $\{c \mid c \in \mathcal{L}\}$.
- The set $\mathcal{E}$ is the given (initial) training set. It consists of positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$, $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$. Let n denote the number of examples in $\mathcal{E}$, $n^\oplus$ of which are positive and $n^\ominus$ negative, $n = n^\oplus + n^\ominus$.
- The hypothesis $\mathcal{H}$ currently built by the learner is called the *current hypothesis*, and the currently built clause c the *current clause*. The current clause c has the form $T \leftarrow Q$. The head T is an atom $p(X_1, \ldots, X_n)$ where p is the target predicate symbol, and the body Q is a conjunction of literals $L_1, \ldots, L_m$.
- The *current training set* $\mathcal{E}_{cur}$ is the set of training examples given to the learner when building the current clause c. It consists of n_{cur} examples, $n^\oplus_{cur}$ positive and $n^\ominus_{cur}$ negative, $n_{cur} = n^\oplus_{cur} + n^\ominus_{cur}$.

---

[2] $\mathcal{P}(\mathcal{L})$ denotes the set of finite subsets of $\mathcal{L}$.

- The *local training set* $\mathcal{E}_c$ is a subset of the current training set $\mathcal{E}_{cur}$ of examples covered by c. Let $n(c)$ denote the number of examples in the local training set, $n^\oplus(c)$ of which are positive and $n^\ominus(c)$ are negative, $n(c) = n^\oplus(c) + n^\ominus(c)$.

A *generic top-down empirical ILP algorithm* can now be outlined. It consists of three main steps:

1. *pre-processing* of the training set,
2. *construction* of a hypothesis, and
3. *post-processing* of the hypothesis.

Pre-processing of training examples

Pre-processing handles missing argument values in training examples and, when no negative examples are given, the generation of negative examples. Different ways of generating negative examples are possible. Most frequently, systems use the so-called generation under the 'closed-world assumption'. In this case, all examples which are not given as positive are generated and labeled negative [Quinlan 1990]. This method is only appropriate in exact, finite domains where all positive examples are in fact given.[3] The user must be most careful when applying the different methods for negative example generation (the 'closed-world assumption', 'partial closed-world assumption' or the 'near-misses' mode) and must at least be able to interpret the results accordingly [Lavrač 1990, Lavrač et al. 1991a].

Construction of a hypothesis

Hypothesis construction assumes the current training set $\mathcal{E}_{cur}$ (initially set to the entire training set $\mathcal{E}$) and the current hypothesis $\mathcal{H}$ (initially set to the empty set of clauses). The algorithm consists of two **repeat** loops, referred to as *covering* and *specialization*. The covering loop performs hypothesis construction and the specialization loop performs clause construction.

---

[3] Even in exact domains the 'closed-world assumption' mode of negative examples generation is sometimes questionable, since most systems assume the same distribution of positive and negative examples in the training and testing set; this assumption can be violated in the case of negative examples generation.

The *covering* loop implements the covering algorithm, known from attribute-value learners AQ [Michalski 1983, Michalski et al. 1986] and CN2 [Clark and Niblett 1989]. It constructs a hypothesis in three main steps:

- construct a clause,
- add the clause to the current hypothesis, and
- remove from the current training set the positive examples covered by the clause.

This loop is typically repeated until all the positive examples are covered.

The main part is clause construction, implemented in the *specialization* loop of the generic ILP algorithm. Clause construction is performed by means of refinement operator ρ which takes the current clause $c = T \leftarrow Q$ and builds a set of its refinements $\rho(c) = \{c' \mid c < c'\}$ by adding a literal L to the body of c. Each refinement c' has the form $c' = T \leftarrow Q'$, where $Q' = Q, L$ (comma stands for conjunction).[4] The specialization loop starts with a clause $c = T \leftarrow$ which has an empty body. In hill-climbing search of the refinement graph, the algorithm keeps one 'best' clause and replaces it with its 'best' refinement at each specialization step, until the necessity stopping criterion is satisfied. In beam search, the algorithm keeps several 'best' clauses instead of one.

Algorithm 3.2 outlines the basic steps of a generic top-down empirical ILP algorithm. It assumes the given (pre-processed) set of training examples $\mathcal{E}$, the language bias $\mathcal{L}$, the definitions of background knowledge predicates $\mathcal{B}$, and the heuristics to be used in clause construction. It also assumes the hill-climbing search of the refinement graph.

The two **repeat** loops are controlled by two *stopping criteria* (also called *quality criteria*).

---

[4] We assume that all specializations of c occur by adding a literal to the body of clause $T \leftarrow Q$, including the specialization in which a variable in the head is replaced by a constant or a structured term. Otherwise, the notation $T' \leftarrow Q'$ should have been used.

Algorithm 3.2 (Generic top-down ILP algorithm)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
Initialize $\mathcal{H} := \emptyset$.
repeat {covering}
  Initialize $c := T \leftarrow$ .
  repeat {specialization}
    Find the best refinement $c_{best} \in \rho(c)$.
    Assign $c := c_{best}$.
  until Necessity stopping criterion is satisfied.
  Add $c$ to $\mathcal{H}$ to get new hypothesis $\mathcal{H}' := \mathcal{H} \cup \{c\}$.
  Remove positive examples covered by $c$ from $\mathcal{E}_{cur}$ to get new
    training set $\mathcal{E}'_{cur} := \mathcal{E}_{cur} - covers(\mathcal{B}, \mathcal{H}', \mathcal{E}^{+}_{cur})$.
  Assign $\mathcal{E}_{cur} := \mathcal{E}'_{cur}, \mathcal{H} := \mathcal{H}'$.
until Sufficiency stopping criterion is satisfied.

Output: Hypothesis $\mathcal{H}$.
```

- In the clause construction (*specialization*) loop, the *necessity stopping criterion* decides when to stop adding literals to a clause.
- In the hypothesis construction (*covering*) loop, the *sufficiency stopping criterion* decides when to stop adding clauses to the hypothesis.

Different stopping criteria are usually applied in domains with perfect (e.g., exact, non-noisy) data and in domains with imperfect (e.g., noisy) data. In domains with perfect data, the necessity criterion requires consistency which means no covered negative examples, and the sufficiency criterion requires completeness which means coverage of all positive examples. In domains with imperfect data, instead of consistency and completeness, heuristic stopping criteria are applied. Consistency and completeness, as well as heuristic stopping criteria, are based on the number of positive and negative examples covered by a clause or hypothesis.

In intensional ILP systems, the set of examples covered by the current clause c is determined with regard to the current hypothesis $\mathcal{H}$, i.e., as $covers(\mathcal{B}, \mathcal{H} \cup \{c\}, \mathcal{E}_{cur})$. This notion is used in the generic top-down empirical ILP algorithm (Algorithm 3.2). Extensional ILP systems, such as FOIL, use the notion of extensional coverage at the appropriate places in the generic algorithm. As can be seen from the above definition of extensional coverage, the set of examples covered by the current clause is independent of the current hypothesis in extensional ILP systems, i.e., is computed as $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}_{cur})$.

When defining the heuristics used in ILP systems, we will take $n^{\oplus}(c)$ and $n^{\ominus}(c)$ to be context dependent, i.e., to take into account the extensional/intensional notion of coverage used in the ILP system. In intensional systems, $n^{\oplus}(c)$ and $n^{\ominus}(c)$ will denote the numbers of positive and negative examples in the set $covers(\mathcal{B}, \mathcal{H} \cup \{c\}, \mathcal{E}_{cur})$. In extensional systems, $n^{\oplus}(c)$ and $n^{\ominus}(c)$ will denote the numbers of positive and negative examples in the set $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}_{cur})$, where $\mathcal{M}$ is a ground model of $\mathcal{B}$.

Different heuristics are discussed in detail in Chapter 8, but are also outlined here for completeness. The simplest heuristic $f(c)$ is the *expected accuracy* of a clause $A(c) = p(\oplus|c)$. The expected accuracy is defined as the probability that an example covered by clause c is positive. *Informativity*, defined as $I(c) = -log_2\, p(\oplus|c)$, is another frequently used heuristic. More elaborate versions of these heuristics include *accuracy gain* $AG(c', c) = A(c') - A(c)$ and *information gain* $IG(c', c) = I(c) - I(c')$, as well as *weighted accuracy gain* $WAG(c', c)$ and *weighted information gain* $WIG(c', c)$. A variant of weighted information gain used in FOIL is described in Section 4.1. Different probability estimates (discussed in detail in Chapter 8) can be used to compute $p(\oplus|c)$, the most frequent (but not the most appropriate) being the *relative frequency* of covered positive examples $p(\oplus|c) = \frac{n^{\oplus}(c)}{n^{\oplus}(c)+n^{\ominus}(c)}$. More appropriate probability estimates are the Laplace estimate and the m-estimate [Cestnik 1990].

Post-processing of a hypothesis

Post-processing of an induced hypothesis $\mathcal{H}$ aims to reduce the complexity of $\mathcal{H}$ and to improve the accuracy of the hypothesis when classifying unseen cases. Note that post-processing of a single clause can also be performed immediately after the specialization loop, which is done in

FOIL (see Algorithm 4.1). Post-processing is performed by removing *irrelevant literals* from a clause and by removing *irrelevant clauses* from the hypothesis. The definitions of the irrelevance of literals/clauses follows.

Irrelevance of literals Let d be the clause obtained by removing literal L from the body of $c \in \mathcal{H}$. Literal L in clause c is *irrelevant* if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^{+}) \subseteq covers(\mathcal{B}, \mathcal{H} \cup \{d\} \setminus \{c\}, \mathcal{E}^{+})$ and $covers(\mathcal{B}, \mathcal{H} \cup \{d\} \setminus \{c\}, \mathcal{E}^{-}) \subseteq covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^{-})$.[5]

Irrelevance of clauses Clause $c \in \mathcal{H}$ is *irrelevant* if

- $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^{+}) \subseteq covers(\mathcal{B}, \mathcal{H} - \{c\}, \mathcal{E}^{+})$, and
- $covers(\mathcal{B}, \mathcal{H} - \{c\}, \mathcal{E}^{-}) \subseteq covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^{-})$.

In perfect (non-noisy, exact) domains, irrelevant literals and irrelevant clauses (as defined above) are eliminated. On the other hand, in imperfect domains heuristic *simplification criteria* are used instead. The heuristics that evaluate the quality of a clause/set of clauses, mentioned in clause construction, can be used also as simplification criteria, the most frequent being the expected accuracy $A(c)$ or $A(\mathcal{H})$.

For practical reasons, instead of discussing various simplification criteria to be used in imperfect domains, we simply redefine the notion of *irrelevance* of literals and clauses using the expected accuracy heuristic. In this case, the definitions of literal/clause irrelevance are the following. A literal in clause c is irrelevant if it can be removed from the body of c without decreasing the expected classification accuracy $A(c)$ of the clause. A clause is irrelevant if its elimination from $\mathcal{H}$ does not decrease the accuracy of the hypothesis. The more formal definitions are stated below.

Irrelevance of literals – imperfect data Let d be the clause obtained from c by removing literal L from the body of c. Literal L in clause c is *irrelevant* if $A(c) \leq A(d)$.

Irrelevance of clauses – imperfect data Clause $c \in \mathcal{H}$ is irrelevant if $A(\mathcal{H}) \leq A(\mathcal{H} - \{c\})$.

---

[5] For extensional ILP systems, this amounts to $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}^{+}) \subseteq covers_{ext}(\mathcal{M}, \{d\}, \mathcal{E}^{+})$ and $covers_{ext}(\mathcal{M}, \{d\}, \mathcal{E}^{-}) \subseteq covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}^{-})$.

A general post-processing mechanism, as used in an ILP system FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], is the *reduced error pruning*, which implements a sophisticated way of removing irrelevant literals/clauses using the expected accuracy estimate. The idea of reduced error pruning is based on Quinlan's mechanism for post-processing of if-then rules generated from decision trees [Quinlan 1987a, 1987b]. The post-processing mechanisms of FOIL and LINUS can be viewed as variants of reduced error pruning.

In Section 1.1, the classification accuracy was mentioned only as one of the success criteria. Other criteria could be used as simplification criteria in post-processing and as stopping criteria in clause construction. Therefore, other approaches should also be mentioned, especially the ones based on information compression [Muggleton and Buntine 1988, Muggleton et al. 1992b] and those based on the statistical significance of a clause (a set of clauses) [Kalbfleish 1979, Clark and Niblett 1989].

Part II

Empirical ILP

4

An overview of empirical ILP systems

This chapter gives an overview of several empirical ILP systems. Special attention is paid to FOIL, which has been the basis for a substantial part of the work in this book. The systems GOLEM and MOBAL are also briefly described. Some other empirical ILP systems are touched upon for completeness.

4.1 An overview of FOIL

FOIL [Quinlan 1990] extends some ideas from the attribute-value learning paradigm to the ILP paradigm. In particular, it uses a covering approach similar to AQ [Michalski 1983] and an information based search heuristic similar to ID3 [Quinlan 1986]. From MIS [Shapiro 1983] it inherits the idea of top-down search of refinement graphs.

The hypothesis language $\mathcal{L}$ in FOIL is restricted to function-free program clauses. Constants and compound terms may not appear in induced clauses. The body of a clause is a conjunction of literals A or *not* A, where A is an atom. Literals in the body have either a predicate symbol q_i from the background knowledge $\mathcal{B}$, or the target predicate symbol, which means that recursive clauses can be induced. At least one of the variables in the arguments of a body literal must appear in the head of the clause or in one of the literals to its left. No literals that bind a variable to a constant are allowed.[1]

Training examples are function-free ground facts (constant tuples). Background knowledge $\mathcal{B}$ consists of extensional predicate definitions

---

[1] Literals that bind a variable to a constant value have the form $X = value$. The lack of this feature could be circumvented in FOIL by introducing additional predicates into $\mathcal{B}$ which have the form $is\_a\_value(X)$ for each possible $value$. This feature has been recently added to FOIL4 [Cameron-Jones and Quinlan 1993].

given by a finite set of function-free ground facts, i.e., $\mathcal{B} = \mathcal{M}$ where $\mathcal{M}$ denotes a ground model.

The FOIL algorithm is basically the same as the generic empirical ILP algorithm from Section 3.2.2. It consists of three basic steps: pre-processing of training examples, hypothesis construction and hypothesis post-processing.

Negative examples are not necessarily given to FOIL. They may be generated in pre-processing, based on the closed-world assumption. Hypothesis construction in FOIL is basically the same as the *covering* loop of Algorithm 3.2. Post-processing of the induced hypothesis is a form of reduced error pruning, briefly mentioned in Section 3.2.2 and described in some more detail in Sections 8.3 and 8.4.

The *covering* loop of the FOIL algorithm (Algorithm 4.1) assumes the given (pre-processed) set of training examples $\mathcal{E}$, the language bias $\mathcal{L}$ of function-free program clauses, the extensional background knowledge $\mathcal{B}$, as well as the search heuristics and the heuristics used as the stopping criteria.

Algorithm 4.1 (FOIL – the covering algorithm)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
Initialize $\mathcal{H} := \emptyset$.
repeat {covering}
   Initialize clause $c := T \leftarrow$ .
   Call the $SpecializationAlgorithm(c, \mathcal{E}_{cur})$
      to find a clause $c_{best}$.
   Assign $c := c_{best}$.
   Post-process $c$ by removing irrelevant literals to get $c'$.
   Add $c'$ to $\mathcal{H}$ to get a new hypothesis $\mathcal{H}' := \mathcal{H} \cup \{c'\}$.
   Remove positive examples covered by $c'$ from $\mathcal{E}_{cur}$ to get
      a new training set $\mathcal{E}'_{cur} := \mathcal{E}_{cur} - covers_{ext}(\mathcal{B}, \{c'\}, \mathcal{E}^+_{cur})$.
   Assign $\mathcal{E}_{cur} := \mathcal{E}'_{cur}$, $\mathcal{H} := \mathcal{H}'$.
until $\mathcal{E}^+_{cur} = \emptyset$ or encoding constraints violated.

Output: Hypothesis $\mathcal{H}$.
```

In the inner *specialization* loop, FOIL seeks a clause of the form

$$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_m$$

Finding a clause consists of a number of refinement steps. Search starts with the clause with an empty body. At each step, the clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$ built so far is refined by adding a literal L_i to its body. Literals are atoms A (or negated atoms *not* A) where A is of the form $X_j = X_s$ or $q_k(Y_1, Y_2, ..., Y_{n_k})$, where the X variables appear in c_i, the Y variables either appear in c_i or are new existentially quantified variables, and q_k is a predicate symbol from the background knowledge $\mathcal{B}$. At least one of the Y variables has to appear in c_i. Recursive calls, i.e., atoms of the form $p(Z_1, Z_2, ..., Z_n)$ may also be added to the body of c_i. This requires at least one irreflexive partial ordering $X_s < Z_s$, $1 \leq s \leq n$, to hold in order to prevent the clause from causing infinite recursion (see Chapter 7). A more sophisticated approach to preventing infinite recursion developed along these lines has been recently implemented within FOIL4 [Cameron-Jones and Quinlan 1993].

In the specialization loop, FOIL makes use of a *local training set* which is initially set to the current training set $\mathcal{E}_1 = \mathcal{E}_{cur}$. While $\mathcal{E}_{cur}$ consists of n-tuples, the local training set consists of m-tuples of constants, each of them being a value assignment to the m variables in the current clause. Let $\mathcal{E}_i$ denote the local training set of tuples that satisfy the current clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$. Let n_i denote the number of tuples in $\mathcal{E}_i$. Elements of $\mathcal{E}_i^+$ are positive tuples and elements of $\mathcal{E}_i^-$ are negative tuples; the numbers of tuples in these sets are denoted by $n_i^{\oplus}$ and $n_i^{\ominus}$, respectively.[2]

The FOIL specialization loop, implementing the function $SpecializationAlgorithm(c, \mathcal{E}_{cur})$, is given in Algorithm 4.2. In each refinement step, clause c_{i+1} is obtained by adding a literal L_i to the body of the clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$, which

---

[2] In the case that $m = n$, the m-tuples of constants in the local training set $\mathcal{E}_i$ consist of those tuples from $\mathcal{E}_{cur}$ that are covered by c_i. Therefore, the number of examples from $\mathcal{E}_{cur}$ covered by c_i could be denoted by $n(c_i)$. However, since the set contains 'extended' tuples (m can be greater than n), the distinction between the local training set $\mathcal{E}_i$ and the set of examples from $\mathcal{E}_{cur}$ covered by c_i needs to be kept and a different notation used.

Algorithm 4.2 (FOIL – the specialization algorithm)

```
Initialize local training set $\mathcal{E}_i := \mathcal{E}_{cur}$.
Initialize current clause $c_i := c$.
Initialize $i := 1$.
while $\mathcal{E}_i^- \neq \emptyset$ or encoding constraints violated do
   Find the best literal $L_i$ to add to the body of $c_i = T \leftarrow Q$
      and construct $c_{i+1} := T \leftarrow Q, L_i$.
   Form a new local training set $\mathcal{E}_{i+1}$ as a set of extensions of
      the tuples in $\mathcal{E}_i$ that satisfy $L_i$.
   Assign $c := c_{i+1}$.
   Increment $i$.
endwhile

Output: Clause $c$.
```

covers the set of tuples $\mathcal{E}_i$. The literal L_i is either an atom A of the form $q_k(Y_1, Y_2, \ldots, Y_{n_k})$ or $X_j = X_s$, or *not* A. Some of the variables $Y_1, Y_2, \ldots, Y_{n_k}$ belong to the 'old' variables already occurring in c_i, $\{OV_1, \ldots, OV_{Old}\}$, while some are 'new', $\{NV_1, \ldots, NV_{New}\}$, i.e., introduced by the literal L_i. The set of tuples $\mathcal{E}_{i+1}$ covered by clause c_{i+1} is the set of ground $(Old + New)$-tuples (instantiations of $< OV_1, \ldots, OV_{Old}, NV_1, \ldots, NV_{New} >$) for which the body $L_1, L_2, \ldots, L_{i-1}, L_i$ is true. Producing a new training set $\mathcal{E}_{i+1}$ as a set of extensions of the tuples in $\mathcal{E}_i$ that can satisfy L_i can be expressed in the relational algebra terminology by saying that $\mathcal{E}_{i+1}$ is the natural join of $\mathcal{E}_i$ with the relation corresponding to literal L_i.

Table 4.1 illustrates the FOIL specialization algorithm on the family relations problem from Section 1.4. The search starts with clause $c_1 = daughter(X, Y) \leftarrow$. The initial local training set $\mathcal{E}_1$ contains all training examples, two of which are positive ($n_1^{\oplus} = 2$) and two negative ($n_1^{\ominus} = 2$). Choosing literal $L_1 = female(X)$ gives rise to the new local training set $\mathcal{E}_2$ of tuples covered by c_2, which contains two positive ($n_2^{\oplus} = 2$) and one negative tuple ($n_2^{\ominus} = 1$). Adding the second literal $L_2 = parent(Y, X)$ to the body of clause c_2 produces clause c_3 with a local training set $\mathcal{E}_3$ containing only positive tuples. Thus, a consistent clause c_3 (which covers no negative tuples) is generated.

Current clause c_1 : $daughter(X, Y) \leftarrow$					
$\mathcal{E}_1$	$(mary, ann)$	$\oplus$		$n_1^{\oplus} = 2$	$I(c_1) = 1.00$
	(eve, tom)	$\oplus$		$n_1^{\ominus} = 2$	
	(tom, ann)	$\ominus$	$L_1 = female(X)$		
	(eve, ann)	$\ominus$	$Gain(L_1) = 0.84$	$n_1^{\oplus\oplus} = 2$	
Current clause c_2 : $daughter(X, Y) \leftarrow female(X)$					
$\mathcal{E}_2$	$(mary, ann)$	$\oplus$		$n_2^{\oplus} = 2$	$I(c_2) = 0.58$
	(eve, tom)	$\oplus$		$n_2^{\ominus} = 1$	
	(eve, ann)	$\ominus$	$L_2 = parent(Y, X)$		
			$Gain(L_2) = 1.16$	$n_2^{\oplus\oplus} = 2$	
Current clause c_3 : $daughter(X, Y) \leftarrow female(X), parent(Y, X)$					
$\mathcal{E}_3$	$(mary, ann)$	$\oplus$		$n_3^{\oplus} = 2$	$I(c_3) = 0.00$
	(eve, tom)	$\oplus$		$n_3^{\ominus} = 0$	

Table 4.1: FOIL trace for the family relation problem.

If a positive literal with new (existentially quantified) variables is added to the body of the clause, the size (arity) of the tuples in the local training set increases. A tuple from $\mathcal{E}_i$ may also give rise to more than one tuple in $\mathcal{E}_{i+1}$. Consider again the problem of learning family relations, with the first literal $L_1 = parent(Y, Z)$ introducing a new variable Z (see Table 4.2). All of the tuples in $\mathcal{E}_1$ have two successors in $\mathcal{E}_2$, which contains eight 3-tuples. The tuples in $\mathcal{E}_{i+1}$ inherit the labels $\oplus$ and $\ominus$ from their parents in $\mathcal{E}_i$.

The choice of literals is directed by an entropy-based search heuristic, called *weighted information gain*, which can be described as follows. If the local set of tuples $\mathcal{E}_i$ contains n_i tuples, $n_i^{\oplus}$ of which are positive and $n_i^{\ominus}$ negative, the information needed to signal that a tuple is positive is $I(c_i) = -log_2(\frac{n_i^{\oplus}}{n_i}) = -log_2(\frac{n_i^{\oplus}}{n_i^{\oplus}+n_i^{\ominus}})$ (see Section 8.4 which explains the informativity and the information gain heuristics in more detail). Let the choice of the next literal L_i give rise to a new tuple set $\mathcal{E}_{i+1}$; the information needed for the same signal is then $I(c_{i+1}) = -log_2(\frac{n_{i+1}^{\oplus}}{n_{i+1}})$. Note that if L_i does not introduce new variables,

Current clause c_1 : $daughter(X,Y) \leftarrow$				
$\mathcal{E}_1$	$(mary, ann)$	$\oplus$		$n_1^{\oplus} = 2$
	(eve, tom)	$\oplus$		$n_1^{\ominus} = 2$
	(tom, ann)	$\ominus$	$L_1 = parent(Y,Z)$	
	(eve, ann)	$\ominus$		$n_1^{\oplus\oplus} = 2$
Current clause c_2 : $daughter(X,Y) \leftarrow parent(Y,Z)$				
$\mathcal{E}_2$	$(mary, ann, mary)$	$\oplus$		$n_2^{\oplus} = 4$
	$(mary, ann, tom)$	$\oplus$		
	(eve, tom, eve)	$\oplus$		
	(eve, tom, ian)	$\oplus$		
	$(tom, ann, mary)$	$\ominus$		$n_2^{\ominus} = 4$
	(tom, ann, tom)	$\ominus$		
	$(eve, ann, mary)$	$\ominus$		
	(eve, ann, tom)	$\ominus$		

Table 4.2: The effect of new variables on the current training set of tuples in FOIL.

then $\mathcal{E}_{i+1} \subseteq \mathcal{E}_i$, $n_{i+1}^{\oplus} \leq n_i^{\oplus}$ and $n_{i+1}^{\ominus} \leq n_i^{\ominus}$. However, if new variables are introduced, it can happen that $n_{i+1}^{\oplus} > n_i^{\oplus}$ and/or $n_{i+1}^{\ominus} > n_i^{\ominus}$. If $n_i^{\oplus\oplus}$ of the positive tuples in $\mathcal{E}_i$ are represented by one or more tuples in $\mathcal{E}_{i+1}$, the information gained by selecting literal L_i amounts to $Gain(L_i) = WIG(c_{i+1}, c_i) = n_i^{\oplus\oplus} \times (I(c_i) - I(c_{i+1}))$. The literal with the highest gain is selected at each step. $WIG(c_{i+1}, c_i)$ stands for weighted information gain, i.e., the *information gain* $I(c_i) - I(c_{i+1})$ *weighted* by the factor $n_i^{\oplus\oplus}$.

For the example, illustrated in Table 4.1, $I(c_1) = -log_2(\frac{2}{2+2}) = 1.00$, $I(c_2) = -log_2(\frac{2}{2+1}) = 0.58$, and $I(c_3) = -log_2(\frac{2}{2+0}) = 0.00$. As $n_1^{\oplus\oplus} = n_2^{\oplus\oplus} = 2$, we have $Gain(L_1) = 2 \times (I(c_1) - I(c_2)) = 2 \times 0.42 = 0.84$, and similarly $Gain(L_2) = 2 \times 0.58 = 1.16$.

The heuristic allows for efficient pruning, which is responsible for the efficiency of FOIL. Let L_i be a positive literal with $Gain(L_i) = n_i^{\oplus\oplus} \times (I(c_i) - I(c_{i+1}))$. L_i may contain new variables and their replacement with old variables can never increase $n_i^{\oplus\oplus}$. Moreover, such replacement can at best produce a set $\mathcal{E}_{i+1}$ containing only $\oplus$ tuples, i.e., with $I(c_{i+1}) = 0$. The maximum gain that can be achieved by any literal obtained by substituting new variables in L_i with old is $MaximumGain(L_i) = n_i^{\oplus\oplus} \times I(c_i)$. If this maximum gain is less than the best gain achieved by a literal so far, no literals that can be obtained with such replacement are considered.

To implement noise-handling, FOIL employs stopping criteria based on the *encoding length restriction*, which limits the number of bits used to encode a clause to the number of bits needed to explicitly indicate the positive examples covered by it. If a clause c_i covers $n^{\oplus}(c_i)$ positive examples out of n_{cur} examples in the current training set $\mathcal{E}_{cur}$, the number of bits used to encode the clause must not exceed $ExplicitBits(c_i, \mathcal{E}_{cur}) = log_2(n_{cur}) + log_2(\binom{n_{cur}}{n^{\oplus}(c_i)})$ bits. The construction of a clause is stopped (the *necessity stopping criterion*) when it covers no negative examples (is consistent) or when no more bits are available for adding literals to its body (see Section 8.4). In the latter case, the clause is retained in the hypothesis if it is more than 85% accurate or is discarded otherwise. The search for clauses stops (the *sufficiency stopping criterion*) when no new clause can be constructed under the encoding length restriction, or alternatively, when all positive examples are covered.

Improvements of FOIL, implemented in FOIL2.0 [Quinlan 1991], include *determinate* literals (an idea borrowed from GOLEM), types and mode declarations of predicates, as well as post-processing of clauses. Consider a partially developed clause which has some old variables and a corresponding set of tuples. Determinate literals [Quinlan 1991] are literals that introduce new variables for which each $\oplus$ tuple has exactly one extension and each $\ominus$ tuple has at most one extension in the new training set (see Section 5.6 for the definition of determinacy). Unless a literal is found with gain close to the maximum, all determinate literals are added to the body of the clause under development. Upon completion of the clause, irrelevant literals are eliminated. Determinate literals alleviate some of the problems in FOIL, which are due to its short-sighted hill-climbing search strategy.

4.2 An overview of GOLEM

Both the training examples $\mathcal{E}$ and the background knowledge $\mathcal{B}$ in GOLEM are restricted to ground facts. If $\mathcal{B}$ is given in the form of non-ground Horn clauses it has to be converted to an *h-easy* ground model $\mathcal{M}$ [Muggleton and Feng 1990]. Unlike in FOIL, terms containing function symbols are allowed to appear in both the examples and the background knowledge.

GOLEM is based on Plotkin's [1969] notion of relative least general generalization (*rlgg*), which in turn is based on the notion of least general generalization (*lgg*) introduced in Section 3.1.1.

Let us recall the definition of *rlgg* from Section 3.1.1. If the background knowledge consists of ground facts, and K represents the conjunction of all these facts, the *rlgg* of two positive training examples e_1 and e_2 relative to the given background knowledge, is defined as follows:

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

Such a clause can contain infinitely many literals or at least grow exponentially with the number of examples. Since such a clause can be intractably large, GOLEM uses some constraints on introducing new variables into the body of the *rlgg*. The variables in the body of the *rlgg* have to be *determinate*, i.e., their values have to be, directly or indirectly, uniquely determined by the values of the variables in the head of the *rlgg* (see Section 5.6 for the definition of determinacy). Even under the determinacy constraint, *rlgg*s are usually quite long clauses which contain superfluous (irrelevant) literals in the body. GOLEM uses negative examples and mode declarations (specifying the input and the output arguments of a predicate) to reduce the size of the clauses. If eliminating a literal from the body does not cause the clause to cover negative examples, then the irrelevant literal is discarded and the reduction procedure is continued with the remaining clause.

The *rlgg* of the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ from the family example of Section 1.4 is computed as follows. For notational convenience, the following abbreviations are used: *d-daughter, p-parent, f-female, a-ann, e-eve, m-mary, t-tom, i-ian.* The conjunction of facts from the background knowledge (comma stands for conjunction) is

$$K = p(a,m), p(a,t), p(t,e), p(t,i), f(a), f(m), f(e)$$

We further have

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

which produces the following clause:

$$\begin{gathered}\underline{d(V_{m,e}, V_{a,t}) \leftarrow} p(a,m), p(a,t), p(t,e), p(t,i), f(a), f(m), f(e),\\ p(a, V_{m,t}), \underline{p(V_{a,t}, V_{m,e}),} p(V_{a,t}, V_{m,i}), p(V_{a,t}, V_{t,e}),\\ p(V_{a,t}, V_{t,i}), p(t, V_{e,i}), f(V_{a,m}), f(V_{a,e}), \underline{f(V_{m,e})}\end{gathered}$$

In the above clause, $V_{x,y}$ stands for $lgg(x,y)$, for each x and y. Eliminating the irrelevant literals yields the following clause:

$$d(V_{m,e}, V_{a,t}) \leftarrow p(V_{a,t}, V_{m,e}), f(V_{m,e})$$

which stands for

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

To generate a single clause, GOLEM first randomly picks several pairs of positive examples, computes their determinate *rlgg*s and chooses the one with greatest coverage (the greatest number of examples covered). This clause is further generalized by randomly choosing new positive examples and computing the *rlgg*s of the clause and each of the examples. Again, among the resulting *rlgg*s the one with the greatest coverage is chosen. The generalization process is repeated until the coverage of the best clause stops increasing. In post-processing, the number of literals is reduced by eliminating irrelevant literals. This means that the clause is further generalized.

To construct definitions consisting of more than one clause, GOLEM uses the covering approach: it constructs a clause covering some positive examples, removes the covered examples from the training set and repeats the whole process. It should be noted that clauses induced by GOLEM may be allowed to cover some pre-determined small number of (noisy) negative examples, thus allowing the constructed definition to be inconsistent. A measure of information compression [Muggleton et al. 1992b] was recently used in conjunction with GOLEM to deal with noisy examples.

4.3 An overview of MOBAL

The system MOBAL [Morik 1991], a further development of BLIP [Morik 1989], is an integrated knowledge acquisition environment consisting of several tools. Some of them are: a *model acquisition tool* which abstracts rule models from rules, a *sort taxonomy tool* which clusters constant terms occurring as arguments in training examples $\mathcal{E}$, and a *predicate structuring tool* which abstracts rule sets to a topology hierarchy. Each of these tools provides information that is used in restricting the search space.

MOBAL contains RDT [Kietz and Wrobel 1992], an empirical ILP learner that learns multiple predicate definitions and uses MOBAL's inference engine to generate new facts (to extend the ground model of the current theory) whenever a new predicate definition is induced. Training examples $\mathcal{E}$ are ground facts, and no function symbols are allowed. The arguments of the examples are typed. The types (sorts) can be input by the user or can be computed automatically by the sort taxonomy tool.

In the focus of our interest are *rule models*, which represent a mechanism for reducing the search space and are used as templates for constructing hypotheses. They are provided by the user or can be derived from input rules by the model acquisition tool. In the process of learning, the use of rule models is further restricted by limitations imposed by the topology and the sort taxonomy.

A *rule model* R has the form $T \leftarrow L_1, \ldots, L_m$ where T and L_i are literal schemas. Each *literal schema* L_i has the form $Q_i(Y_1, \ldots, Y_{n_i})$ or *not* $Q_j(Y_1, \ldots, Y_{n_j})$ where all non-predicate variables Y_k are implicitly universally quantified. The *predicate variable* Q_i can be instantiated only to a predicate symbol of the specified arity n_i.

The search for hypotheses in RDT is guided by rule models. The hypothesis space that is actually searched is the set of all predicate ground instantiations of rule models with predicates from the background knowledge $\mathcal{B}$. An *instantiation* Θ is a second-order substitution, i.e., a finite set of pairs P/p where P is a predicate variable and p is a predicate symbol. An instantiation of a rule model R (denoted by $R\Theta$) is obtained by replacing each predicate variable in R by the corresponding predicate symbol from the set Θ. A second-order rule model R is *predicate ground* if all of its predicate variables are instantiated; in this case, R becomes a first-order rule.

For learning the *daughter* relationship from the family relations problem of Section 1.4 the user also needs to define (among other rule models) the model:

$$P(X,Y) \leftarrow R(X), Q(Y,X)$$

In this case, the correct clause

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

can be induced. Otherwise, RDT would in vain try to find this clause. The above model could also be instantiated to the clause

$$son(X,Y) \leftarrow male(X), parent(Y,X)$$

We can thus see that one rule model can be useful for solving several learning problems.

The space of hypotheses, limited by rule models, is still very large. The hierarchical order of rule models based on extended θ-subsumption allows the system to prune parts of the hypothesis space. For each rule model, all possible instantiations of predicate variables are tried systematically and are tested against the ground facts in $\mathcal{E}$. An instantiation is possible if the predicates that substitute predicate variables have a compatible arity and correspond to the sort and topology limitations. The topology hierarchy states which predicates are to be used in a model. Sorts represent a syntactic way of expressing semantics; for each argument of each predicate a set of possible values is computed. When instantiating the rule model, a substitution of a predicate variable is valid only if the sorts of arguments correspond to sorts of arguments of an already instantiated predicate variable. RDT can also use constants in the induced hypotheses, but the rule models have to be designed accordingly.

4.4 Other empirical ILP systems

Early empirical systems that touched the problem of learning relational descriptions are INDUCE [Michalski 1983] and ML-SMART [Bergadano and Giordana 1988, 1990]. They are both multiple predicate learners, but only for the case of nullary predicates (i.e., predicates corresponding to propositional classes). ML-SMART also has an explanation-based learning capability.

Many empirical ILP systems, based on the FOIL approach, have emerged recently. These include CHAM [Kijsirikul et al. 1991], FORTE [Richards and Mooney 1991], RX [Tangkitvanich and Shimura 1992] and FOCL [Pazzani and Kibler 1992, Pazzani et al. 1991].

CHAM was initially designed to overcome the problems of FOIL caused by hill-climbing in the case of non-discriminating and non-determinate literals. For that purpose it introduces a similarity-based heuristic, called likelihood, which is used in addition to the information gain heuristic. Other differences to FOIL include a seed-based beam search, similar to the one in AQ. A further extension of CHAM, called CHAMP [Kijsirikul et al. 1992], effectively addresses the problem of inventing new predicates, i.e., representation change during the learning process. However, the issue of representation change in the presence of noise is left unresolved.

FOCL, FORTE and RX belong to a family of ILP learners that could be named non-interactive theory revisors. Given an initial definition of the target concept, and an additional set of training examples which disagree with the initial definition, the task is to construct a new definition, based on the old one, which will agree with all the training examples. In the limit, when no initial target concept definition is given, they behave very similar to FOIL. Otherwise, they attempt to detect and repair incorrect clauses or, if positive examples are not covered, to induce clauses that will cover them.

FOCL and RX integrate ILP and explanation-based learning. FOCL uses the information-gain metric from FOIL as a universal measure to decide whether to operationalize existing parts of the definition or inductively construct new parts of the definition. RX also operationalizes the existing theory and constructs new operational parts, but in addition de-operationalizes the constructed definition in the end. Of the above systems, only FOCL is able to handle noisy data [Brunk and Pazzani 1991]. After constructing a consistent and complete definition, a *reduced error pruning* procedure is applied to it. An outline of this procedure can be found in Section 8.4.

The empirical ILP system MARKUS [Grobelnik 1992] is a variant of MIS [Shapiro 1983]. It also belongs to the class of empirical theory revisors, as it can take an initial theory and correct/complete it. The main differences from MIS include: the optimal generation of the refinement graph (without duplicates), the use of iterative deepening instead of breadth-first search, the removal of irrelevant clauses, and the use of negation.

A recent empirical ILP system addressing the problem of learning multiple predicates is MPL (Multiple Predicate Learner [De Raedt et al. 1993a, 1993b]). Although it might seem easy and natural to apply a single predicate learner several times in order to generate definitions of several predicates, there are serious problems with this approach, e.g., when learning mutually recursive predicates. The problems stem from the order dependency of the single predicate learning tasks and the use of the extensional definition of coverage in most empirical ILP systems. Based on ideas from FOIL, MPL overcomes these problems and is able to solve multiple predicate learning problems beyond the scope of other empirical ILP systems.

5

LINUS: Using attribute-value learners in an ILP framework

This chapter presents a method for effectively using background knowledge in learning both propositional and relational descriptions. The method, implemented in the system LINUS, employs propositional learners in a more expressive logic programming framework. LINUS is an ILP system integrating several attribute-value learners. It can be viewed as an ILP toolkit of learning algorithms out of which one or more can be selected in order to find the best solution to a given problem. The chapter first describes three attribute-value learners which are part of the LINUS environment. The method of using background knowledge in learning is described both for attribute-value and ILP problems. The LINUS algorithm, implementing this method, is then outlined, followed by a description of the system's language bias of constrained deductive hierarchical database (DHDB) clauses. Next, the computational complexity of the LINUS learning task is addressed. Finally, an extension to the original LINUS algorithm, called DINUS (Determinate LINUS), is proposed which allows for extending the language bias of constrained DHDB clauses to determinate clauses.

5.1 An outline of the LINUS environment

LINUS is an ILP learner which induces hypotheses in the form of constrained deductive hierarchical database (DHDB) clauses. As input it takes training examples $\mathcal{E}$, given as ground facts, and background knowledge $\mathcal{B}$ in the form of (possibly recursive) deductive database (DDB) clauses. The definitions of DDB and DHDB clauses can be found in Section 2.1.

The main idea in LINUS is to transform the problem of learning relational DHDB descriptions into a propositional learning task [Lavrač et al. 1991a]. This is achieved by the so-called *DHDB interface*, consisting of over 2000 lines of Prolog code. At present, LINUS incorporates the propositional learners ASSISTANT [Cestnik et al. 1987], NEWGEM [Mozetič 1985b] and CN2 [Clark and Boswell 1991]. It is easy to incorporate other propositional learners, out of which one or more can be selected to be applied.

The interface transforms the training examples from the DHDB form into the form of attribute-value tuples. The most important feature of this interface is that, by taking into account the types of the arguments of the target predicate, applications of background predicates and functions are considered as attributes for attribute-value learning. Existing attribute-value learners can then be used to induce if-then rules. Finally, the induced if-then rules are transformed into the form of DHDB clauses by the DHDB interface. The scheme of the LINUS environment is given in Figure 5.1.

LINUS is an integrated environment for learning both attribute and relational descriptions using attribute-value learners. The LINUS algorithm is a descendant of the learning algorithm used in QuMAS (Qualitative Model Acquisition System [Mozetič 1987]) which was used to learn functions of components of a qualitative model of the heart in the KARDIO expert system for diagnosing cardiac arrhythmias [Bratko et al. 1989].

LINUS was successfully applied to the problem of learning diagnostic rules in rheumatology, described in Chapter 11 [Lavrač et al. 1991b, 1993], the problem of learning relational descriptions in several domains known from the machine learning literature [Lavrač et al. 1991a], described also in Chapter 6, and the problem of learning rules for finite element mesh design in CAD [Džeroski 1991], described also in Chapter 12. Using propositional attribute-value learning algorithms, LINUS allows for recent advances in handling imperfect data in these algorithms to be applied easily to real-life, imperfect data. Experiments in a chess endgame domain with a controlled amount of noise, described in Chapter 10 [Džeroski and Lavrač 1991b, Lavrač and Džeroski 1992b], show that LINUS performs well on imperfect, noisy data of relational nature.

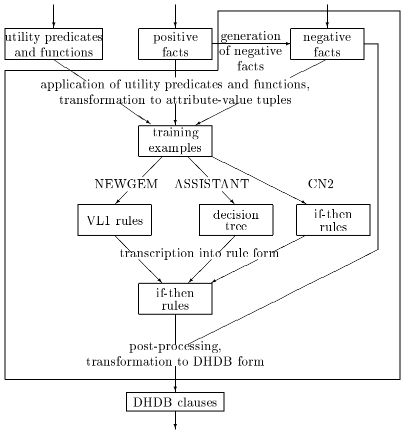

Figure 5.1: An overview of LINUS.

5.2 Attribute-value learning

LINUS incorporates several attribute-value learners into the logic programming environment. Let us first recall the propositional learning setting: examples are tuples of attribute values labeled with a concept name (or labeled $\oplus$ for the positive instances of the concept and $\ominus$ for the negative instances), and the induced hypotheses typically have the form of if-then rules or decision trees. This section illustrates a simple propositional learning task and gives an outline of three propositional learners incorporated into the LINUS environment.

5.2.1 An example learning problem

Suppose that the learning task is to find a description of friendly and unfriendly robots [Wnek et al. 1990] from a set of examples described by six attributes. In our simplified problem, given in Table 5.1, six examples are given, described by five attributes of three different types. Attributes $Smiling$ and Has_tie have values of the type $yes\_no = \{yes, no\}$, attribute $Holding$ has values of the type $holding = \{sword, balloon, flag\}$, and attributes $Head\_shape$ and $Body\_shape$ have values of the type $shape = \{round, square, octagon\}$.

From the examples in Table 5.1, an algorithm from the AQ family induces the following if-then rules:

	Attributes and values				
Class	*Smiling*	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*
friendly	*yes*	*balloon*	*yes*	*square*	*square*
friendly	*yes*	*flag*	*yes*	*octagon*	*octagon*
unfriendly	*yes*	*sword*	*yes*	*round*	*octagon*
unfriendly	*yes*	*sword*	*no*	*square*	*octagon*
unfriendly	*no*	*sword*	*no*	*octagon*	*round*
unfriendly	*no*	*flag*	*no*	*round*	*octagon*

Table 5.1: Examples of friendly and unfriendly robots.

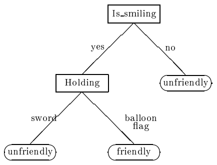

Figure 5.2: Decision tree for the robot world.

$$\begin{array}{lll} Class = friendly & \textbf{if} & [Smiling = yes] \wedge \\ & & [Holding = balloon \vee flag] \\ Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

Using an ID3-based decision tree building algorithm ASSISTANT, the tree in Figure 5.2 is induced. If transcribed into the form of rules, the tree produces exactly the above if-then rules.

5.2.2 ASSISTANT

ASSISTANT is a descendant of CLS (Concept Learning System [Hunt et al. 1966]) and ID3 [Quinlan 1979, 1986]. The ID3 algorithm generates a hypothesis in the form of a decision tree that enables the classification of new objects. As such, it is a member of the TDIDT (Top-Down Induction of Decision Trees [Quinlan 1986]) family of inductive learning programs.

CLS and ID3 implement a simple mechanism for generating a decision tree from a given set of attribute-value tuples. Each of the interior nodes of the tree is labeled by an attribute, while branches that lead from the node are labeled by the possible values of the attribute. The tree construction process is heuristically guided by choosing the 'most informative' attribute at each step, aimed at minimizing the expected number of tests needed for classification.

Let $\mathcal{E}$ be the set of training examples, and $C_1, \ldots C_N$ the decision classes. ID3 constructs a decision tree by repeatedly calling Algorithm 5.1 in each generated node of the tree.

Algorithm 5.1 (ID3)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
if All the examples in $\mathcal{E}_{cur}$ are of the same class $C_j$
then Generate a leaf labeled $C_j$.
else
  Select the 'most informative' attribute $A$ with values
    $\{v_1, \ldots, v_n\}$ for the root of the tree.
  Split $\mathcal{E}_{cur}$ into subsets $\mathcal{E}_1, \ldots, \mathcal{E}_n$ according to the values
    $v_1, \ldots, v_n$ of attribute $A$.
  for $i = 1$ to $1 = n$ do
    Recursively build a subtree $T_i$ for $\mathcal{E}_i$.
  endfor

Output: Decision tree $T$ with the root $A$ and subtrees
  $T_1, \ldots, T_n$ as shown in Figure 5.3.
```

Ideally, each leaf is labeled by exactly one class name. However, leaves can also be empty (NULL leaves), or can be labeled by more than one class name (SEARCH leaves). If there are no training examples having attribute values that would lead to a particular leaf then this is

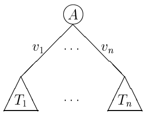

Figure 5.3: Decision tree built by ID3 with attribute A in the root.

a NULL leaf. SEARCH leaves appear when there are identical training examples with different class names.

At each step of the ID3 algorithm, the search heuristic chooses the 'most informative' attribute based on an information-theoretic measure called *entropy*. Let $E(\mathcal{E})$ be the *entropy* of the training set $\mathcal{E}$, i.e., the amount of information needed for the classification of one object [Shanon and Weaver 1964]:

$$E(\mathcal{E}) = -\sum_{j=1}^{N} p_j \cdot \log_2 p_j \tag{5.1}$$

where p_j is the prior probability of class C_j, $j \in \{1, \ldots, N\}$. Usually, p_j is computed as the relative frequency of examples from C_j in $\mathcal{E}$, i.e., $p_j = \frac{n^{C_j}}{n}$, where $n = |\mathcal{E}|$ and n^{C_j} is the number of examples from $\mathcal{E}$ of class C_j. Note that other (better) probability estimates can be used instead of the relative frequency (see Section 8.6 which gives an overview of probability estimates).

The entropy of the training set $\mathcal{E}$, after splitting it according to the values $v_1, \ldots, v_n$ of attribute A equals:

$$E(\mathcal{E}|A) = -\sum_{i=1}^{n} p_i \cdot \sum_{j=1}^{N} \frac{p_{ij}}{p_i} \cdot \log_2 \frac{p_{ij}}{p_i} \tag{5.2}$$

where p_i is the prior probability that A has value v_i, and p_{ij} is the prior probability that an example belonging to class C_j has value v_i of attribute A.

The *informativity* of attribute A is a measure aimed at minimizing the number of tests needed for the classification of a new object. It is defined as:

$$I(A) = E(\mathcal{E}) - E(\mathcal{E}|A) \tag{5.3}$$

At each step of the algorithm, the most informative attribute is the one which maximizes $I(A)$, or equivalently, minimizes $E(\mathcal{E}|A)$.

In ASSISTANT [Cestnik et al. 1987], the basic ID3 learning algorithm is extended in several ways which include: handling incompletely specified training examples, binarization of continuous attributes, binary construction of decision trees, pruning of unreliable parts of the tree and plausible classification based on the 'naive' Bayesian principle.

One of the most important features is tree pruning, used as a mechanism for handling noisy data. Tree pruning is aimed at producing trees which do not overfit possibly erroneous data. In tree pruning, the unreliable parts of a tree are eliminated in order to increase the classification accuracy of the tree on unseen cases.

There are two types of pruning:

- *pre-pruning*, performed during the construction of a decision tree, which decides whether to stop or to continue tree construction at a certain node, and
- *post-pruning*, employed after the decision tree has been constructed, which estimates the expected classification errors in the nodes of the tree and then decides whether to prune the subtrees rooted at those nodes.

The pruning techniques are based on the heuristic called the expected classification accuracy, or alternatively, the expected classification error estimate, mentioned in Section 3.2.2 and described in more detail in Section 8.4. Experiments in different domains have shown that the features enhancing ID3, as implemented in ASSISTANT, generate decision trees which are smaller, simpler and more comprehensible, and at the same time more accurate when classifying new objects [Cestnik et al. 1987, Cestnik and Bratko 1991].

5.2.3 NEWGEM

NEWGEM [Mozetič 1985b] is a descendant of the AQ7-AQ11 series of learning systems [Michalski 1969, 1983] and was further developed into AQ15 [Michalski et al. 1986] and AQ17 [Bloedorn and Michalski 1991].

Both AQ and NEWGEM induce hypotheses in the form of if-then rules. Descriptions are learned separately for each individual class. Objects from a given class are considered its positive examples, and all other objects are negative examples of the concept. Training examples are described as tuples of attribute values. Attributes can be of three types: nominal, linear or structured. Concept descriptions in the form of if-then rules are represented in the VL1 notation (Variable-valued Logic 1 [Michalski 1974]).

In the VL1 notation, a *selector* relates an attribute to a value or to a disjunction of values, e.g., $[Smiling = yes]$ and $[Holding = balloon \vee flag]$, respectively. A *complex* is a conjunction of selectors, e.g., $[Smiling = yes] \wedge [Holding = balloon \vee flag]$. A hypothesis is a set of if-then rules of the form

$$Class = ClassValue \ \ \textbf{if} \ \ Cover$$

where $ClassValue$ denotes one of the decision classes, and $Cover$ is a a disjunction of complexes covering all positive examples and none of the negative examples of the class. NEWGEM induces the following hypothesis from the examples given in Table 5.1.

$$\begin{array}{lll} Class = friendly & \textbf{if} & [Smiling = yes] \wedge [Holding = balloon \vee flag] \\ Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

In order to make the terminology more uniform with the rest of the book, we will use the term *condition* instead of selector, the term *body* instead of complex, and a rule with a disjunction of two complexes

$$\begin{array}{lll} Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

will be written in the form of two rules

$$\begin{array}{lll} Class = unfriendly & \textbf{if} & [Smiling = no] \\ Class = unfriendly & \textbf{if} & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

It should be noted that **if** denotes a relation which is not necessarily implication. Whether the interpretation of **if** as implication is correct depends on the training data [Mozetič and Lavrač 1988]. In the original VL1 notation the symbol $\Leftarrow$ is used; the direction of the arrow ($\Leftarrow$ or $\Rightarrow$) just indicates the natural direction of inference when a rule of this form is used.

The AQ algorithm first introduced the so-called 'covering' approach. As mentioned earlier, in the case of N classes $C_1, \ldots C_N$, the task of learning the descriptions of N classes is first transformed into N learning tasks, one for each individual class C_i. For each task, examples of class C_i form $\mathcal{E}^+$ and examples belonging to all other classes form $\mathcal{E}^-$. Given the current training set of examples $\mathcal{E}_{cur}$ (initially set to the entire training set) and the current hypothesis $\mathcal{H}$ (which is most frequently initially set to the empty set of rules), the 'covering' loop of the NEWGEM algorithm constructs a hypothesis in three main steps:

- construct a rule,
- add the rule to the current hypothesis, and
- remove from the current training set the positive examples covered by the rule.

This loop is repeated until all the positive examples are covered. Algorithm 5.2 implements this loop.

Algorithm 5.2 (NEWGEM – the covering algorithm)

```
Initialize $\mathcal{H} := \emptyset$.
Initialize $\mathcal{E}_{cur} := \mathcal{E}^+ \cup \mathcal{E}^-$.
repeat
  Select an uncovered positive example $Seed$.
  Call the $BeamSearchAlgorithm(\mathcal{E}_{cur}, Seed, ClassValue)$
    to find the best body $BestBody$.
  $\mathcal{H} := \mathcal{H} \cup \{Class = ClassValue$ if $BestBody\}$,
    i.e., add the best rule to $\mathcal{H}$.
  Remove from $\mathcal{E}_{cur}$
    the positive examples covered by $BestBody$.
until $\mathcal{E}^+_{cur} = \emptyset$, i.e., $\mathcal{H}$ is complete.

Output: Hypothesis $\mathcal{H}$.
```

AQ starts hypothesis construction from an empty hypothesis or, alternatively, from an initial hypothesis supplied by the user. It generates a hypothesis in steps, each step producing one rule of the hypothesis. The algorithm selects a positive example (called a *Seed*) and starts with the maximally general rule (one with all possible values for all attributes). The body of the rule is then specialized so that it does not cover any negative example, but still covers the *Seed*. The algorithm keeps several alternative rules which maximize the number of covered positive examples and are as simple as possible. The set of alternative rule bodies is called a beam (a *star* in the AQ terminology). When no negative example is covered, the algorithm selects the best rule and adds it to the current hypothesis. The algorithm repeats the procedure until all positive examples are covered.

The procedure for generating a rule (body) to be added to the current hypothesis, invoked in the second step of the **repeat** loop of Algorithm 5.2, is given in Algorithm 5.3.

Algorithm 5.3 (NEWGEM – the beam search algorithm)

```
Initialize $Beam := \{true\}$.
while Any $Body$ in $Beam$ covers negative examples do
  Select a covered negative example.
  Specialize all inconsistent $Bodies$ to uncover the negative
    example, but still cover the $Seed$.
  Keep only the best $BeamSize$ bodies in $Beam$
    i.e., remove the worst bodies if $BeamSize$ is exceeded.
endwhile
Select $BestBody$ from $Beam$
  according to user-defined preference criteria.

Output: $BestBody$.
```

In the initial maximally general rule body, all attributes have all possible values. *Specialization* of a rule body with respect to the *Seed* (that has to be covered) and a negative example (that should not be covered) is done by removing some values from an internal disjunction of values of individual attributes so that the rule remains as general as possible. Notice that alternative specializations are possible which causes the combinatorial complexity of the algorithm.

The *measure of rule quality* determines a partial order of alternative rules and is a parameter (LEF – Lexical Evaluation Function) defined by the user. This is an essential part of the inductive bias used by the system. The measure consists of a list of criteria that are applied to each rule. When the first criterion cannot discriminate between two rules, the second one is used, and so on. The following is a list of default criteria that order rules from the best to worse:

- higher number of covered positive examples,
- lower number of conditions in the rule,
- lower total number of values in internal disjunctions.

The second user-defined parameter (called *BeamSize*) is the number of alternative rules to be considered by the algorithm, i.e., the width of the beam in beam search. Default value for this parameter is 10.

The AQ algorithm enables the generation of rules of different degrees of generality. Rules may be *general* (having the minimal number of attributes, each with the maximal number of disjunctive values), *minimal* (the minimal number of both attributes and values), or *specific* (the maximal number of attributes, each with minimal number of values).

When learning from inconsistent examples, i.e., examples having the same attribute values but belonging to different classes, the AQ algorithm provides three options:

- inconsistent examples are treated as positive examples for each class,
- inconsistent examples are treated as negative examples for each class,
- inconsistent examples are removed from the data.

NEWGEM and AQ15 have the incremental learning facility. The user can supply an initial hypothesis. The programs implement the method of learning with *full memory* (as opposed to learning with *partial memory*). In this type of learning, the program remembers all the training examples seen so far, as well as the rules it formed. New rules are guaranteed to be consistent and complete with respect to all (old and new) training examples [Reinke and Michalski 1988].

5.2.4 CN2

The rule induction system CN2 [Clark and Niblett 1989] combines the ability to cope with noisy data of the algorithms of the TDIDT family with the if-then rule form and the flexible search strategy of the AQ family. It has retained the covering approach and the beam search mechanism from AQ, but has removed its dependency on specific examples (*Seed*) during the search. Furthermore, in order to deal with noisy data, it has extended the search space to rules that do not perform perfectly on the training data. Initially, CN2 used an entropy-based search heuristic to induce ordered rules [Clark and Niblett 1989]. It has recently been extended with the ability to induce unordered rules and to use Bayesian accuracy estimates as search heuristics [Clark and Boswell 1991, Džeroski et al. 1992a]. The heuristics used for handling noisy data are described in detail in Section 8.3.

When learning if-then rules, objects of a given class are labeled $\oplus$ and treated as positive examples $\mathcal{E}^+$, and all other objects are treated as negative examples $\mathcal{E}^-$ of the selected class. Given the current training set of examples $\mathcal{E}_{cur}$ (initially set to the entire training set) and the current hypothesis $\mathcal{H}$ (initially set to the empty set of rules), the outer loop of the CN2 algorithm [Clark and Boswell 1991] for inducing unordered rules (Algorithm 5.4) implements the 'covering' algorithm outlined in Section 5.2.3.

Algorithm 5.4 is an outline of the CN2 covering algorithm, working for a particular class $ClassValue$.

The best body is chosen according to the search heuristic which measures the expected classification accuracy of the rule, estimated by the Laplace probability estimate [Clark and Boswell 1991].

Let N be the total number of classes in the problem. Let $n^{\oplus}(c)$ be the number of positive examples of the class $ClassValue$ and $n(c)$ the total number of examples in the current training set $\mathcal{E}_{cur}$, covered by a rule c. The Laplace classification accuracy estimate $A(c)$ estimates the probability that an example covered by rule c is positive as:

$$A(c) = p(\oplus|c) = \frac{n^{\oplus}(c)+1}{n(c)+N}$$

In its previous version, instead of the Laplace estimate, CN2 was using an entropy-based heuristic [Clark and Niblett 1989].

Algorithm 5.4 (CN2 – the covering algorithm)

```
Initialize $\mathcal{H} := \emptyset$.
Initialize $\mathcal{E}_{cur} := \mathcal{E}^+ \cup \mathcal{E}^-$.
repeat
  Call the $BeamSearchAlgorithm(\mathcal{E}_{cur}, ClassValue)$ to find
    the $BestBody$ of a rule.
  if $BestBody \neq not\_found$
  then
    Add the best rule to $\mathcal{H}$, i.e.,
      $\mathcal{H} := \mathcal{H} \cup \{Class = ClassValue$ if $BestBody\}$.
    Remove from $\mathcal{E}_{cur}$
      the positive examples covered by $BestBody$.
until $BestBody = not\_found$.

Output: Hypothesis $\mathcal{H}$.
```

The beam search for the best body of a rule in Algorithm 5.5 proceeds in a top-down fashion. The search starts with the most general body ($true$), which covers all the training examples. At each step, a set of candidates ($Beam$) for the best body, as well as the best body found so far are kept. To specialize a body, a condition of the form $X = a$ is added to it as an additional conjunct. All the possible specializations of the candidates from $Beam$ are considered. The best body found so far is accordingly updated and the $BeamSize$ most promising candidates among the newly generated specializations are chosen.

The best body found so far is, in addition, tested for its statistical significance. This is to ensure that it represents a genuine regularity in the training examples and not a regularity which is due to chance. In this way, the undesirable bias of the entropy and apparent accuracy metrics used to estimate the quality of a clause is, at least partly, avoided. For a detailed treatment of the heuristics in CN2 and their role in handling noisy data, we refer the reader to Section 8.3.

Algorithm 5.5 (CN2 – the beam search algorithm)

```
Initialize $Beam := \{true\}$.
Initialize $NewBeam := \emptyset$.
Initialize $BestBody := not\_found$.
while $Beam \neq \emptyset$ do
  for each $Body$ in $Beam$ do
    for each possible specialization $Spec$ of $Body$ do
      if $Spec$ is better than $BestBody$ and
        $Spec$ is statistically significant
      then
        $BestBody := Spec$.
        Add $Spec$ to $NewBeam$.
      if Size of $NewBeam > BeamSize$
      then remove worst body from $NewBeam$.
    endfor
  endfor
  $Beam := NewBeam$.
endwhile

Output: $BestBody$.
```

5.3 Using background knowledge in learning

This section shows how background knowledge can be effectively applied to induce compact hypotheses in both the attribute-value and the ILP setting.

5.3.1 Using background knowledge in attribute-value learning

The method for using background knowledge in attribute-value learning is outlined on the robots example from Section 5.2.1. It is shown that the induced hypothesis can include new terms, given as background knowledge about the world of robots. Background knowledge can be expressed in the form of *functions* of attribute values or *relations* among attribute values. In learning attribute descriptions, these functions/relations give rise to new attributes which are considered in the learning process. If the background knowledge is represented in the form of functions, the value of a new attribute is computed as a function of the values of existing attributes, in turn for each training example. The range of values for the function is either a finite set of discrete values or an interval of real numbers. On the other hand, if the background knowledge has the form of relations, the only values the new attribute can have are $true$ and $false$ (if the values of the corresponding attributes of the example do/do not satisfy the relation). In other words, relations are boolean functions.

For example, background knowledge can check for the equality of attribute values for pairs of attributes of the same type (i.e., attributes with the same set of values). In the world of robots from Section 5.2.1 this would lead to two new attributes that test the equalities $Smiling = Has\_tie$ and $Head\_shape = Body\_shape$. For simplicity, let us consider only the new attribute $Head\_shape = Body\_shape$, named $Same\_shape$, the values of which are $true$ and $false$. Using this idea, initially introduced by Mozetič [1987], an extended set of tuples is generated and used in learning. The set of attribute-value tuples for the world of robots is given in Table 5.2, where f stands for the class $friendly$ and u for $unfriendly$.

Since its two values $true$ and $false$ completely distinguish between the friendly and unfriendly robots, the new attribute $Same\_shape$ is the only attribute in the decision tree induced by ASSISTANT. The tree is shown in Figure 5.4.

	Attributes and values					
Class	*Smiling*	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*	*Same_shape*
f	yes	$balloon$	yes	$square$	$square$	$true$
f	yes	$flag$	yes	$octagon$	$octagon$	$true$
u	yes	$sword$	yes	$round$	$octagon$	$false$
u	yes	$sword$	no	$square$	$octagon$	$false$
u	no	$sword$	no	$octagon$	$round$	$false$
u	no	$flag$	no	$round$	$octagon$	$false$

Table 5.2: Examples of friendly and unfriendly robots. The last column gives the values of the new attribute $Same\_shape$.

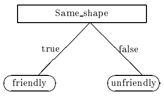

Figure 5.4: Decision tree built by using background knowledge.

If the examples from Table 5.2 are given to NEWGEM, the following if-then rules are induced:

$$
\begin{array}{lll}
Class = friendly & \textbf{if} & [Same\_shape = true] \\
Class = unfriendly & \textbf{if} & [Same\_shape = false]
\end{array}
$$

The example shows that new attributes, expressing functions of (relations among) the original attributes that describe the examples, can be more informative than the original attributes. Using background knowledge, a learner can build more compact hypotheses, which can potentially have a better classification accuracy on unseen cases.

5.3.2 Transforming ILP problems to propositional form

The method outlined in Section 5.3.1 is based on the idea that the use of background knowledge can introduce new attributes for learning. This same method can also be used in the ILP framework. To do so, the learning problem is transformed from relational to attribute-value form and solved by an attribute-value learner. This approach is feasible only for a restricted class of ILP problems.

The algorithm which solves ILP problems by transforming them into propositional form consists of the following three steps:

- The learning problem is transformed from relational to attribute-value form.
- The transformed learning problem is solved by an attribute-value learner.
- The induced hypothesis is transformed back into relational form.

In this section, the hypothesis language is restricted to function-free program clauses which are typed, constrained and non-recursive, i.e., to function-free constrained DHDB clauses (see the definitions in Section 2.1). The above algorithm allows for a variety of approaches developed for propositional problems, including noise-handling techniques in attribute-value algorithms, such as ASSISTANT [Cestnik et al. 1987] or CN2 [Clark and Niblett 1989], to be used for learning relations.

The algorithm is illustrated on a simple ILP problem of learning family relationships, similar to the one in Section 1.4. The task is to define the target relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations $female$, $male$ and $parent$. All the variables are of type $person$. The type $person$ is defined as $person = \{ann, eve, pat, sue, tom\}$. There are two positive and two negative examples of the target relation. The training examples and the relations from the background knowledge are given in Table 5.3, which is similar to Table 1.1.

The first step of the algorithm, i.e., the transformation of the ILP problem into attribute-value form, is performed as follows. The possible applications of the background predicates on the arguments of the target relation are determined, taking into account argument types. Each such application introduces a new attribute. In our example, all variables are of the same type $person$. The corresponding attribute-value learning problem is given in Table 5.4, where f stands for $female$, m for $male$ and p for $parent$. The attribute-value tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation.

In Table 5.4, *variables* stand for the arguments of the target relation, and *propositional features* denote the newly constructed attributes of the propositional learning task. When learning function-free DHDB clauses, only the new attributes are considered for learning. If we remove the function-free restriction, the arguments of the target relation are used as attributes in the propositional task as well (see Section 5.4.4 for a more detailed explanation).

Training examples		*Background knowledge*		
$daughter(sue, eve).$	$\oplus$	$parent(eve, sue).$	$female(ann).$	$male(pat).$
$daughter(ann, pat).$	$\oplus$	$parent(ann, tom).$	$female(sue).$	$male(tom).$
$daughter(tom, ann).$	$\ominus$	$parent(pat, ann).$	$female(eve).$	
$daughter(eve, ann).$	$\ominus$	$parent(tom, sue).$		

Table 5.3: Another simple family relationships problem.

	Variables		*Propositional features*							
C	X	Y	$f(X)$	$f(Y)$	$m(X)$	$m(Y)$	$p(X,X)$	$p(X,Y)$	$p(Y,X)$	$p(Y,Y)$
$\oplus$	sue	eve	$true$	$true$	$false$	$false$	$false$	$false$	$true$	$false$
$\oplus$	ann	pat	$true$	$false$	$false$	$true$	$false$	$false$	$true$	$false$
$\ominus$	tom	ann	$false$	$true$	$true$	$false$	$false$	$false$	$true$	$false$
$\ominus$	eve	ann	$true$	$true$	$false$	$false$	$false$	$false$	$false$	$false$

Table 5.4: Propositional form of the *daughter* relationship problem.

In the second step, an attribute-value learning program induces the following if-then rule from the tuples in Table 5.4:

$$Class = \oplus \quad \textbf{if} \quad [female(X) = true] \ \wedge \ [parent(Y, X) = true]$$

In the last step, the induced if-then rules are transformed into DHDB clauses. In our example, we get the following clause:

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

Note that the same result can be obtained on a similar learning problem, where the target relation $daughter(X, Y)$ is to be defined in terms of the relations $female$, $male$ and $parent$, given the background knowledge from Table 5.5. It illustrates that in this approach intensional background knowledge in the form of non-ground clauses can be used in addition to ground facts.

5.4 The LINUS algorithm

The method outlined in Section 5.3.2 is implemented in the ILP learner LINUS. Given the training examples $\mathcal{E}$ as ground facts defining the predicate p/n, the definitions of background knowledge predicates q_i/n_i and functions f_j/n_j in $\mathcal{B}$, the types of arguments of predicates p/n and q_i/n_i and functions f_j/n_j, and the language bias $\mathcal{L}$ of constrained DHDB clauses, the LINUS learning algorithm works as follows:

Background knowledge			
$mother(eve, sue)$.	$parent(X, Y) \leftarrow$	$female(ann)$.	$male(pat)$.
$mother(ann, tom)$.	$mother(X, Y)$.	$female(sue)$.	$male(tom)$.
$father(pat, ann)$.	$parent(X, Y) \leftarrow$	$female(eve)$.	
$father(tom, sue)$.	$father(X, Y)$.		

Table 5.5: Intensional background knowledge for learning the *daughter* relationship.

Algorithm 5.6 (LINUS – learning constrained DHDB clauses)

1. Pre-process the training set to establish the sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of positive and negative examples.
2. Using background knowledge $\mathcal{B}$, transform the facts from $\mathcal{E}^+$ and $\mathcal{E}^-$ from the DHDB form into attribute-value *Tuples*.
3. From *Tuples* induce if-then *Rules* by an attribute-value learner.
4. Transform *Rules* into DHDB *Clauses*.
5. Post-process *Clauses* to generate hypothesis $\mathcal{H}$.

Output: Hypothesis $\mathcal{H}$.

5.4.1 Pre-processing of training examples

Pre-processing of the training set involves negative example generation when negative examples are not given. It also involves the handling of missing argument values, described in Section 8.2.

The generation of negative facts in the first step of the LINUS algorithm takes into account the type theory and the 'closed-world assumption', which is a frequent assumption in different logic programming applications. LINUS provides for four options of negative example generation, an idea borrowed from QuMAS [Mozetič 1987].

- Negative facts can be given *explicitly*.

- When generating negative facts under the *closed-world assumption* (the *cwa* mode), all possible combinations of values of the n arguments of the target predicate are generated.

- Under the *partial closed-world assumption* (the *pcwa* mode), for a given combination of values of the n_{Ind} independent variables, all the combinations of values of the n_{Dep} dependent variables are generated.

- In the *near_misses* mode, facts are generated by varying only the value of one of the n variables at a time, where $n = n_{Dep} + n_{Ind}$.

The generated facts are used as negative instances of the target relation, except for those given as positive. Here we have to note that LINUS distinguishes between the so-called *dependent* and *independent* variables. Among the n arguments of the target predicate, n_{Dep} variables are treated as *dependent*, which is analogous to a *class* variable in attribute-value learning. The other n_{Ind} arguments are considered *independent* variables, analogous to *attributes* in attribute-value learning. This distinction is important, since LINUS treats dependent and independent variables differently in its two learning modes: learning definitions of relations (the *relation* learning mode) and learning the descriptions of individual classes (the *class* learning mode). Note that there can be more than one dependent variable in LINUS. In this case, a class value is a combination of values of all dependent variables.

In noisy and inexact domains, LINUS has to be run in the *class* learning mode or in the *relation* learning mode with *explicitly* given negative examples. The other three modes of negative examples generation are not appropriate. When one can not assume that the given training set is complete, the *cwa* negative examples generation mode is obviously not appropriate, as well as the *near_misses* mode. However, in some domains the *pcwa* mode could be used; then one has to be aware of the assumption that a given positive example belonging to one particular class does not belong to any other class.

In non-noisy exact domains, the *cwa* mode can be used. However, this mode is inappropriately used in two cases. The first case is when the set of positive examples is not exhaustive. The second case deals with the problem of representing the training examples since negative examples are generated in the 'object' space and not in the 'relation' space (see comments in Section 6.3).

5.4.2 Post-processing

As mentioned in Section 3.2.2, post-processing eliminates irrelevant literals from the clauses constituting a hypothesis in order to make the induced hypothesis more compact and more accurate when classifying new cases. In non-noisy exact domains a literal L in clause c is irrelevant if the clause c', obtained by eliminating L from c, does not cover any negative examples not covered by c.

In noisy domains, a literal L in a clause c is irrelevant if it can be removed from the body of the clause without decreasing the classification accuracy of the clause on the training set. Algorithm 5.7 eliminates irrelevant literals from a clause.

Algorithm 5.7 (Elimination of irrelevant literals)

```
Initialize $c' := c$.
repeat
  if $A(c) \leq A(c')$ then $c := c'$.
  Compute accuracy estimate $A(c)$ of clause $c$.
  for each literal $L$ in clause $c$ do
    Generate new clause $c_L$
      by eliminating literal $L$ from the body of $c$.
    Compute the accuracy estimate $A(c_L)$.
    if $A(c') \leq A(c_L)$ then $c' := c_L$.
  endfor
until $c = c'$
```

If a clause covers $n^{\oplus}(c)$ examples of the correct class and $n^{\ominus}(c)$ examples of incorrect classes, its expected classification accuracy is estimated in LINUS by the formula proposed in Quinlan [1987a]:

$$A(c) = \frac{n^{\oplus}(c) - 0.5}{n(c)}$$

where $n(c) = n^{\oplus}(c) + n^{\ominus}(c)$. This estimate was used in the experiments of Chapter 11. This is just one of the possible formulas for computing the expected accuracy $A(c)$. Recall that other (better) accuracy estimates, outlined in Chapter 8, could be used instead, e.g., the Laplace estimate [Niblett and Bratko 1986] and the m-estimate [Cestnik 1990].

5.4.3 An example run of LINUS

Let us illustrate in more detail the work of LINUS on a simple example. Consider the problem of learning illegal positions on a chess board with only two pieces, white king and black king. The target relation $illegal(WKf, WKr, BKf, BKr)$ states that the position in which the white king is at (WKf, WKr) and the black king at (BKf, BKr) is illegal. Arguments WKf and BKf are of type *file* (with values a to h), while WKr and BKr are of type $rank$ (with values 1 to 8). Background knowledge is represented by two symmetric predicates $adj\_file(X, Y)$ and $adj\_rank(X, Y)$ which can be applied on arguments of type $file$ and $rank$, respectively (the abbreviation adj stands for $adjacent$). The built-in symmetric predicate equality $X = Y$, which works on arguments of the same type, may also be used in the induced clauses.

A description of the individual steps of the algorithm follows.

1. First, in data pre-processing, the sets of positive and negative facts are established. In our domain, given are one positive example (illegal endgame position, labeled $\oplus$) and two negative examples (legal endgame positions, labeled $\ominus$):

$$\begin{array}{l}
illegal(WKf, WKr, BKf, BKr). \\
\quad illegal(a, 6, a, 7). \quad \oplus \\
\quad illegal(f, 5, c, 4). \quad \ominus \\
\quad illegal(b, 7, b, 3). \quad \ominus
\end{array}$$

2. The given facts are transformed into attribute-value tuples. The algorithm first determines the possible applications of the background predicates on the arguments of the target relation, taking into account argument types. Each such application is considered as an attribute.

In our example, the set of attributes determining the form of the tuples is the following:

⟨ *WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩

Since predicates adj_file and adj_rank are symmetric, their other possible applications $adj\_file(BKf, WKf)$ and $adj\_rank(BKr, WKr)$ are not considered as attributes for learning.

It should be noted that the arguments of the target predicate, WKf, WKr, BKf and BKr, may be included in the selected set of attributes for learning, so that the form of the tuples would be as follows: ⟨ *WKf, WKr, BKf, BKr, WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩. However, knowing that that the illegality of a chess position does not depend on the particular positions of the pieces, but rather their relative positions, we have excluded these arguments from the transformed tuples. In addition, in our experiments, this facilitates the comparison with FOIL, the concept description language of which does not allow for literals of the type $X = value$.

The tuples, i.e., the values of the attributes, are generated by calling the corresponding predicates with argument values from the ground facts of predicate *illegal*. In this case, the attributes can take values *true* or *false*. For the given examples, the following tuples are generated:

⟨ *WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩

$$\begin{array}{ll} \langle true, false, false, true \rangle & \oplus \\ \langle false, false, false, true \rangle & \oplus \\ \langle true, false, false, false \rangle & \ominus \end{array}$$

These tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation. For example, the first tuple actually has the following meaning:

$$\begin{array}{lll} \forall A \forall B \forall C \forall D : & & \\ illegal(A, B, C, D) & \textbf{if} & A = C \wedge not\ (B = D) \wedge \\ & & not\ adj\_file(A, C) \wedge adj\_rank(B, D). \end{array}$$

3. Next, an attribute-value learning program is used to induce a set of if-then rules. When ASSISTANT is used, the induced decision trees are transcribed into the if-then rule form. NEWGEM induces the

following rule from the above three tuples:

$$Class = \oplus \ \ \textbf{if} \ \ (WKf = BKf) = true \wedge adj\_rank(WKr, BKr) = true$$

4. Finally, the induced if-then rules for $Class = \oplus$ are transformed into DHDB clauses. In our example, we get the following clause:

$$\begin{array}{lll} illegal(WKf, WKr, BKf, BKr) & \leftarrow & WKf = BKf, \\ & & adj\_rank(WKr, BKr). \end{array}$$

5. In post-processing, no irrelevant literal is found and the hypothesis remains unchanged.

In summary, the learning problem is transformed from a relational to an attribute-value form and solved by an attribute-value learner. The induced hypothesis is then transformed back into relational form.

5.4.4 Language bias in LINUS

Before giving the restrictions imposed by the hypothesis language $\mathcal{L}$ of deductive hierarchical database clauses in LINUS, it is important to consider the form of training examples $\mathcal{E}$ and, in particular, predicates in $\mathcal{B}$ which actually determine the hypothesis space. Training examples have the form of ground facts (which may contain structured, but non-recursive terms) and background knowledge has the form of deductive database clauses (possibly recursive). Variables are typed.

Background knowledge

In LINUS, predicate definitions in the background knowledge $\mathcal{B}$ are of two types:

Utility functions Utility functions f_j/n_j are annotated predicates; mode declarations specify the *input* and *output* arguments, similar to mode declarations in GOLEM and FOIL2.0 [Quinlan 1991]. When applied to ground input arguments from the training examples, utility functions compute the unique ground values of their output arguments. When used in an induced clause, output variables must be bound to constants.

Utility predicates Utility predicates q_i/n_i have only *input* arguments and can be regarded as boolean utility functions having values $true$ or $false$ only.

A reduction of the hypothesis space is achieved by taking into account the pre-specified types of predicate arguments and by exploiting the fact that some utility predicates are symmetric:

Symmetric predicates Utility predicates can be declared *symmetric* in a set of arguments of the same type. For example, a binary predicate $q_i(X, Y)$ is symmetric in $\{X, Y\}$ if X and Y are of the same type, and $q_i(X, Y) = q_i(Y, X)$ for every value of X and Y. A built-in symmetric utility predicate *equality (=/2)* is defined by default on arguments of the same type.

Note that the complexity analysis in Section 5.5 (equation (5.6)) does not take into account the symmetry of predicates and the use of utility functions f_j.

Hypothesis language

In the current implementation of LINUS, the selected hypothesis language $\mathcal{L}$ is restricted to constrained deductive hierarchical database (DHDB) clauses. In DHDB, variables are typed and recursive predicate definitions are not allowed. In addition, all variables that appear in the body of a clause have to appear in the head as well, i.e., only constrained clauses are induced.

To be specific, the body of an induced clause in LINUS is a conjunction of literals, each having one of the following four forms:

1. a binding of a variable to a value, e.g., $X = a$;
2. an equality of pairs of variables, e.g., $X = Y$;
3. an atom with a predicate symbol (utility predicate) and input arguments which are variables occurring in the head of the clause, e.g., $q_i(X, Y)$; and
4. an atom with a predicate symbol (utility function) having as input arguments variables which occur in the head of the clause, and

output arguments with an instantiated (computed) variable value, e.g., $f_j(X, Y, Z), Z = a$.

In the above, X and Y are variables from the head of the clause, and a is a constant of the appropriate type. Literals of form (2) and (3) can be either positive or negative. Literals of the form $X = a$ under items (1) and (4) may also have the form $X > a$ and/or $X < a$, where a is a real-valued constant.

The attributes given to propositional learners are (1) the arguments of the target predicate, (2)–(3) binary-valued attributes resulting from applications of utility predicates and (4) output arguments of utility functions. Attributes under (1) and (4) may be either discrete or real-valued. For cases (2) and (3) an attribute-value learner will use conditions of the form $A = true$ or $A = false$ in the induced rules, where A is an attribute (cf. examples in Sections 5.3.2 and 5.4.3). These are transcribed to literals A and *not* A, respectively, in the DHDB clauses. For case (1) an attribute-value system will use conditions of the form $X = a$, $X > a$ or $X < a$, which can be immediately used in DHDB clauses. For case (4), in addition to the conditions $Z = a$, $Z > a$ and $Z < a$, the literal $f_j(X, Y, Z)$ has to be added to the DHDB clause so that the value of Z can be computed from the arguments of the target predicate.

To guide induction, LINUS uses meta-level knowledge which can exclude any of the above four cases, thus reducing the search space. For example, if only case (1) is retained, the hypothesis language is restricted to an attribute-value language. This can be achieved by using only the arguments of the target relation as attributes for learning. Cases (2)–(4), on the other hand, result from the use of predicates/functions from $\mathcal{B}$ as attributes for learning. The *equality* predicate *(=/2)*, which generates literals of the form (2), is built-in in LINUS.

The complexity of learning with LINUS is analyzed in Section 5.5 and shows the number of new attributes leading to literals of form (1)–(3) above. A detailed analysis of the actual hypothesis space, including the number of attributes which result in literals of form (4), can be found in Chapter 7 [Džeroski and Lavrač 1992].

Language biases for two typical learning tasks

As mentioned above, meta-level knowledge can be used to adjust the language bias in LINUS, which is done by the user according to the type of learning task at hand. Below we list two types of tasks with the appropriate settings of the language bias:

Class learning mode As an attribute-value learner, LINUS is used to induce descriptions of individual classes. In this case, the classes can be different from $\oplus$ and $\ominus$ and are determined by the values of a selected argument (or a set of arguments) of the target relation. The induced clauses have the form

$$Class = ClassValue \ \ \mathbf{if} \ \ L_1, \ldots, L_m$$

where $ClassValue$ is a class name and literals L_i can take any of the four forms outlined in Section 5.4.4. In the experiments in learning of medical diagnostic rules, described in Chapter 11, LINUS was run in the class learning mode.

Relation learning mode As an ILP learner, LINUS induces constrained DHDB clauses of the form

$$p(X_1, \ldots, X_n) \leftarrow L_1, \ldots, L_m$$

where p is the name of the target predicate and literals L_i have any of the forms (1)–(4) above. In the experiments with LINUS described in Chapter 6, LINUS was used in the relation learning mode, learning function-free clauses; thus, only literals of form (2) and (3) were actually used.

To summarize, the language bias in LINUS is declarative. The user can set the language bias to any of the above forms, depending on the learning task at hand.

5.5 The complexity of learning constrained DHDB clauses

To analyze the complexity of the ILP learning task obtained by using background knowledge in the learning process, let us consider the number of attributes in the transformed learning task [Lavrač et al. 1991a].

Strongly related to this issue is the search complexity in LINUS estimated by the branching factor of the refinement graph, discussed in Section 7.4.2. In this section, only background knowledge predicates q_i/n_i are considered in the analysis, whereas in Section 7.4.2 utility functions f_j/n_j are considered as well. We will illustrate the complexity formulae on the task of learning family relationships used in Section 5.3.1.

The total number of attributes k_{Attr} to be considered by a propositional learner equals

$$k_{Attr} = k_{Arg} + \sum_{i=1}^{l} k_{New,q_i} \tag{5.4}$$

where $k_{Arg} = n$ is the number of arguments of the target relation p (i.e., *variables* in Table 5.4), and k_{New,q_i} is the number of new attributes resulting from the possible applications of the background predicate q_i on the arguments of the target relation.

Under the function-free restriction, k_{Attr} is equal to the total number of new attributes resulting only from the possible applications of the l background predicates on the arguments of the target relation (i.e., *propositional features* in Table 5.4):

$$k_{Attr} = \sum_{i=1}^{l} k_{New,q_i} \tag{5.5}$$

Suppose that u is the number of distinct types of arguments of the target predicate p, u_i is the number of distinct types of arguments of the background predicate q_i, $n_{i,s}$ is the number of arguments of q_i that are of type $\mathcal{T}_s$ and k_{ArgT_s} is the number of arguments of target predicate p that are of type $\mathcal{T}_s$. Then k_{New,q_i} is computed by the following formula:

$$k_{New,q_i} = \prod_{s=1}^{u_i} (k_{ArgT_s})^{n_{i,s}} \tag{5.6}$$

The $n_{i,s}$ places for arguments of type $\mathcal{T}_s$ can be, namely, filled in $(k_{ArgT_s})^{n_{i,s}}$ ways independently from choosing the arguments of q_i which are of different types.

Since a relation where two arguments are identical can be represented by a relation of a smaller arity, the arguments of a background

predicate can be restricted to be distinct. In this case the following formula holds:

$$k_{New,q_i} = \prod_{s=1}^{u_i} \binom{k_{ArgT_s}}{n_{i,s}} \cdot n_{i,s}! \tag{5.7}$$

To illustrate the above formulae on the simple example from Tables 5.3 and 5.4, observe that $q_1 = female$, $q_2 = male$ and $q_3 = parent$. As all arguments are of the same type $\mathcal{T}_1$ ($person$), $u_1 = u_2 = u_3 = 1$ and $k_{ArgT_1} = 2$. Since there is only one type, n_i can be used instead of $n_{i,s}$. In this notation, $n_1 = n_2 = 1$ and $n_3 = 2$. Thus, according to equation (5.6), $k_{New,q_1} = k_{New,q_2} = (k_{ArgT_1})^{n_1} = (k_{ArgT_1})^{n_2} = 2^1 = 2$. This means that there are two applications of each of the predicates $female$ and $male$, namely $female(X)$, $female(Y)$ and $male(X)$, $male(Y)$, respectively. Similarly, $k_{New,q_3} = (k_{ArgT_1})^{n_3} = 2^2 = 4$, the four applications of $parent/2$ being $parent(X, X)$, $parent(X, Y)$, $parent(Y, X)$ and $parent(Y, Y)$. Finally, $k_{Attr} = k_{New,q_1} + k_{New,q_2} + k_{New,q_3} = 2 + 2 + 4 = 8$.

If the built-in background predicate *equality (=/2)* is used, the number of its possible applications equals to:

$$k_{New,=} = \sum_{s=1}^{u} \binom{k_{ArgT_s}}{2} = \sum_{s=1}^{u} \frac{k_{ArgT_s} \cdot (k_{ArgT_s} - 1)}{2}$$

where k_{ArgT_s} denotes the number of arguments of the target predicate that are of the same type $\mathcal{T}_s$, and u is the number of distinct types of arguments of the target relation; from the arguments of type $\mathcal{T}_s$, $\binom{k_{ArgT_s}}{2}$ different attributes of the form $X = Y$ can be constructed.

Assuming a constant upper bound j on the arity of background knowledge predicates, the largest number of attributes that can be obtained in case that all variables are of the same type is of the order $\mathcal{O}(ln^j)$, where l is the number of predicates in the background knowledge and n is the arity of the target predicate. In other words, the size of the transformed learning task is polynomial in the arity n of the target predicate p and the number l of background knowledge predicates. This allows us to use PAC-learnability results for the propositional case in the ILP framework [Džeroski et al. 1992c].

5.6 Weakening the language bias

It was shown [Džeroski et al. 1992c, Lavrač and Džeroski 1992a] that the language bias in LINUS can be weakened, so that LINUS can learn clauses which introduce new variables. To allow for more expressiveness, an idea from GOLEM is borrowed, also used in FOIL2.0 [Quinlan 1991] to alleviate the problems with greedy search. While LINUS allows only 'old' variables from the head of a clause, the idea of determinacy allows for a restricted form of 'new' variables to be introduced into the learned clauses. The very type of restriction (determinacy) allows the LINUS transformation approach to be used in this case as well [Džeroski et al. 1992c, Lavrač and Džeroski 1992a].

By extending the transformation approach of the LINUS algorithm, the same class of logic programs can be learned as induced by GOLEM, namely the class of *determinate logic programs* defined below. In fact, the class of programs induced by the extended LINUS algorithm is slightly larger, as negated literals are allowed in the bodies of clauses. Based on this approach, some positive learnability results were proven for the class of determinate logic programs [Džeroski et al. 1992c] within the PAC-learnability framework of [Valiant 1984] and [Li and Vitányi 1991]. In this section, the language bias of i-determinate clauses is introduced and illustrated on an example. Without loss of generality [Džeroski et al. 1992c, Rouveirol 1992], we will assume a function-free hypothesis language.

5.6.1 The i-determinacy bias

Assuming that an integer constant j is given, only the class of ILP problems where all predicates in $\mathcal{B}$ are of arity at most j will be considered. The notion of *variable depth* is first defined.

Variable depth Consider a clause

$$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_r, ...$$

Variables that appear in the head of the clause have *depth* zero. Let a variable V appear first in literal L_r. Let d be the maximum depth of the other variables in L_r that appear in the clause

$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_{r-1}$. Then the *depth* of variable V is $d+1$.

By setting a maximum variable depth i, the syntactic complexity of clauses in the hypothesis language is restricted.

The following definition of *determinacy*, adapted to the case of function-free clauses, is taken from Džeroski et al. [1992c].

Determinacy A predicate definition is *determinate* if all of its clauses are determinate. A clause is *determinate* if each of its literals is determinate. A literal is *determinate* if each of its variables that do not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals.

i-determinacy The determinacy restriction is called *i-determinacy* for a given maximum variable depth i.

Given the arity bound j, *i-determinacy* implies *ij-determination* as defined by Muggleton and Feng [1990]. Note that while i and j restrict the syntactic complexity of clauses, the notion of determinacy is essentially a semantic restriction, as it depends on the given training examples and background knowledge.

If we set the hypothesis language to the language of i-determinate clauses, the transformation approach of LINUS is still applicable, as demonstrated in Section 5.7. Determinate literals are a natural extension of the utility functions used presently in LINUS. In the light of the above definitions we can say that, in the current implementation of LINUS, only variables of depth 0 and 1 are allowed and that variables of depth 1 may not be used in other literals, except as described in Section 5.4.4.

The arity bound j is fixed. On the other hand, the parameter i can be increased to increase the expressiveness of the hypothesis language. For a fixed j, the user can consider the following series of languages:

$$\mathcal{L}_0 \text{ - } \mathcal{L}_1 \text{ - } \mathcal{L}_2 \text{ - } \ldots$$

where the expressiveness of $\mathcal{L}_i$ (and thus the complexity of the search space) grows with i. The language $\mathcal{L}_0$ is the language of constrained

function-free DHDB clauses. If a solution for the ILP problem cannot be found in the selected language, the next language in the series may be chosen, and the learning process repeated until a complete and consistent solution is found. Along these lines, LINUS could, in principle, shift its bias dynamically, similarly to CLINT [De Raedt 1992] and NINA [Adé and Bruynooghe 1992].

However, there are several problems with shifting bias dynamically. First of all, the series of languages considered has to be in some sense complete. Consider, for example, the case where the target definition is not determinate (is not in any of the languages in the series). In this case the system may continue to shift its bias forever without finding a satisfactory solution. Next, the learning process is repeated for each language in the series, up to the appropriate one. This leads to less efficient learning, although some improvements are possible. The most important problem, however, is when and how to shift the bias if learning data is imperfect, which can easily be the case in empirical ILP. Therefore, we will rather assume a fixed language bias $\mathcal{L}_i$.

5.6.2 An example determinate definition

Let us illustrate the notion of i-determinacy on a simple ILP problem. The task is to define the predicate *grandmother*, where $grandmother(X, Y)$ states that person X is grandmother of person Y, in terms of the background knowledge predicates *father* and *mother*. The training examples are given in Table 5.6. and background knowledge in Table 5.7.

Training examples	
$grandmother(ann, bob).$	⊕
$grandmother(ann, sue).$	⊕
$grandmother(bob, sue).$	⊖
$grandmother(tom, bob).$	⊖

Table 5.6: Training examples for learning the *grandmother* relationship.

Background knowledge		
$father(zak, tom).$	$father(pat, ann).$	$father(zak, jim).$
$mother(ann, tom).$	$mother(liz, ann).$	$mother(ann, jim).$
$father(tom, sue).$	$father(tom, bob).$	$father(jim, dave).$
$mother(eve, sue).$	$mother(eve, bob).$	$mother(jean, dave).$

Table 5.7: Background knowledge for learning the *grandmother* relationship.

A correct target predicate definition is:

$$grandmother(X, Y) \leftarrow father(Z, Y), mother(X, Z).$$
$$grandmother(X, Y) \leftarrow mother(U, Y), mother(X, U).$$

This hypothesis can be induced in a language which is at least as expressive as $\mathcal{L}_1$. The clauses are determinate (but not constrained), because each occurrence of a new variable (i.e., Z in $father(Z, Y)$ and U in $mother(U, Y)$) has only one possible binding given particular values of the other (old) variables in the literal (i.e., Y in this case); this is the case since each person has exactly one mother and father. The hypothesis is function-free and the maximum depth of any variable is one ($i = 1$). The arity bound j has to be at least two to allow induction of the above definition.

However, the logically equivalent hypothesis

$$grandmother(X, Y) \leftarrow mother(X, Z), father(Z, Y).$$
$$grandmother(X, Y) \leftarrow mother(X, U), mother(U, Y).$$

is not determinate, since the new variable Z in the literal $mother(X, Z)$ can have more than one binding for a fixed value of X (e.g., $X = ann$, $Z = tom$ or $Z = jim$); each person can namely have several children. We can thus see that the property of determinacy of a clause does not depend only on the given training examples and background knowledge, but also on the ordering of literals in the body of a clause.

5.7 Learning determinate clauses with DINUS

In this section, LINUS is treated exclusively as an ILP learner using only background knowledge in the form of predicates which may have

both input and output arguments. We show how to weaken the language bias in LINUS and learn determinate DDB clauses. An algorithm that transforms the problem of learning non-recursive function-free i-determinate clauses into a propositional form, called DINUS (Determinate LINUS), is first presented and then illustrated using the example from Section 5.6.2. The back transformation of induced propositional concept descriptions to the program clause form is then briefly described and illustrated by an example. Finally, we discuss how the algorithm can be extended to learn recursive clauses. Dealing with non-recursive clauses is easier, as the recursive case may require querying the user (oracle) in order to complete the transformation process.

5.7.1 Learning non-recursive determinate DDB clauses

For simplicity, we first consider the problem of learning non-recursive clauses. The transformation of the ILP problem of constructing an i-determinate definition for target predicate $p(X_1, X_2, ..., X_n)$ from examples $\mathcal{E}$ and from predicate definitions in $\mathcal{B}$ is described below.

Like the original LINUS algorithm, the extended algorithm also consists of three main steps (the first step contains the most important differences to the the original LINUS algorithm, described in Sections 5.3.2 and 5.4). Note that, for simplicity, pre-processing of training examples and post-processing of hypotheses are not discussed.

- Transform the ILP problem to propositional form.
    1. Construct a list L of determinate literals that introduce new variables of depth at most i. Construct a list V_i of old variables and new variables introduced by determinate literals from L.
    2. Construct a list F of all literals that use predicates from the background knowledge and variables from V_i. The determinate literals from L are excluded from this list, as they do not discriminate between positive and negative examples (since the new variables have a unique binding for each example, due to the determinacy restriction). The resulting list is the list of features (attributes) used for propositional learning.

  3. Transform the examples to propositional form. For each example, the truth value of each of the propositional features is determined by calls to the background knowledge $\mathcal{B}$.

- Apply a propositional learning algorithm to induce a propositional concept description.
- Transform the induced propositional description to a set of determinate DDB clauses, adding the necessary determinate literals which introduced the new variables.

DINUS – The transformation algorithm

Algorithm 5.8 describes the transformation of an ILP problem into propositional form, i.e., the first step of the above algorithm for learning non-recursive determinate clauses. The algorithm assumes that the learning task is to find a predicate definition as a set of determinate DDB clauses with $p(X_1, X_2, \ldots, X_n)$ in the head. Given are the training examples $\mathcal{E}$ as ground facts of the target predicate p/n, the language bias $\mathcal{L}_i$ of i-determinate DDB clauses, the definitions of background knowledge predicates q_s/n_s in $\mathcal{B}$, and the types of arguments of predicates p/n and q_s/n_s. The output of the algorithm is a set of examples $\mathcal{E}_f$ for attribute-value learning in the form of tuples f labeled $\oplus$ for $p(a_1, a_2, ..., a_n) \in \mathcal{E}^+$ and labeled $\ominus$ for $p(a_1, a_2, ..., a_n) \in \mathcal{E}^-$.

The knowledge base $\mathcal{B}$ of background predicates may take the form of a (normal) logic program. In step (3) of Algorithm 5.8 two types of queries have to be posed to this knowledge base.

- The first type are *existential* queries, which are used to determine the values of the new variables. Given a partially instantiated goal (literal containing variables that do not appear in previous literals), an existential query returns the set (possibly empty) of all bindings for the unbound variables which satisfy the literal. For example, the query $mother(X, A)$, where $X = ann$ and A is a new variable would return the set of answers $\{A = tom, A = jim\}$. Note that for determinate literals the set of answers is always a singleton.
- The other type of queries are ground (*membership*) queries about background knowledge predicates, where the goal is completely bound. These are used to determine the truth values of the propositional features.

Algorithm 5.8 (DINUS – the transformation algorithm)

```
Initialize the list of variables $V_0 := \{X_1, X_2, ..., X_n\}$.
Initialize the list of literals $L := \emptyset$.
Initialize the set of tuples $\mathcal{E}_f := \emptyset$.
1. for $r = 1$ to $i$ do
     $D_r := \{q_s(Y_1, Y_2, ..., Y_{j_s}) \mid q_s \in \mathcal{B},\ Y_1, Y_2, ..., Y_{j_s}$ are of the
       appropriate types, $q_s(Y_1, Y_2, ..., Y_{j_s})$ is determinate and contains at least one new variable not in $V_{r-1}$ $\}$.
     $L := L \cup D_r$.
     $V_r := V_{r-1} \cup \{Y \mid Y$ appears in a literal from $D_r\}$.
   endfor
2. $F := \{q_s(Y_1, Y_2, ..., Y_{j_s}) \mid q_s \in \mathcal{B},\ Y_1, Y_2, ..., Y_{j_s} \in V_i\} - L$.
3. for each $p(a_1, a_2, ..., a_n) \in \mathcal{E}$ do
     Determine the values of variables in $V_i$ by executing the body
       of the clause $p(X_1, X_2, ..., X_n) \leftarrow L$ with variables $X_1, ..., X_n$
       bound to $a_1, ..., a_n$.
     Given the values of variables in $V_i$, determine the tuple $f$ of
       truth values of literals in $F$, by querying the background
       knowledge $\mathcal{B}$.
     $\mathcal{E}_f := \mathcal{E}_f \cup \{f\}$.
   endfor
```

In an actual implementation of Algorithm 5.8, steps (1) and (3) should be interleaved, i.e., the values of the new variables and the propositional features should be calculated for each example as they are introduced. In this way, the determinacy of literals in step (1) is automatically tested when the existential query determining the values of the new variables in a literal is posed. A literal is determinate if the set of answers to the existential query is a singleton for all the examples. If the set of answers for some example is not a singleton, the literal is not determinate.

The DINUS algorithm at work: An example

For the grandmother example, given in Tables 5.6 and 5.7 of Section 5.6.2, we have $i = 1$, $l = 2$, and $n = 2$. Let $j \geq 2$. The literal $father(X, A)$, where X is an old and A is a new variable, is not determinate (e.g., $X = tom$, $A = sue$ or $A = bob$). However, if A is old and X is new, the literal is determinate. As the target predicate is $grandmother(X, Y)$, we have $V_0 = \{X, Y\}$ and $D_1 = \{f(U, X), f(V, Y), m(W, X), m(Z, Y)\}$, where f and m stand for $father$ and $mother$, respectively.

This gives $L = D_1$, $V_1 = \{X, Y, U, V, W, Z\}$. The list F includes literals such as $f(X, X)$, $f(X, Y)$, $f(Z, Y)$, $f(W, X)$ and similarly $m(Z, Z)$, $m(V, Y)$, $m(W, W)$, $m(U, X)$, $m(X, V)$, $m(X, Z)$. In fact, the pairs of arguments of f and m are all the pairs from the Cartesian product $V_1 \times V_1$, excluding the pairs that produce literals from D_1.

The transformation process for the ILP problem as defined by the training examples from Table 5.6 and background knowledge from Table 5.7 is illustrated in Table 5.8. For the two positive and the two negative examples of $g(X, Y)$ (g stands for $grandmother$), given are the values of the old variables X and Y, the values of the new variables U, V, W and Z introduced by the determinate literals from the list L, and the values of two of the propositional features from the list F, namely $m(X, V)$ and $m(X, Z)$, denoted by A_1 and A_2, respectively. Note that only the propositional features (attributes $A_1, A_2, \ldots$) are actually considered in the attribute-value learning task.

	Variables		*New variables*				*Propositional features*			
$g(X,Y)$	X	Y	$f(U,X)$	$f(V,Y)$	$m(W,X)$	$m(Z,Y)$	...	$m(X,V)$	$m(X,Z)$	...
Class	X	Y	U	V	W	Z		A_1	A_2	
$\oplus$	*ann*	*bob*	*pat*	*tom*	*liz*	*eve*		*true*	*false*	
$\oplus$	*ann*	*sue*	*pat*	*tom*	*liz*	*eve*		*true*	*false*	
$\ominus$	*bob*	*sue*	*tom*	*tom*	*eve*	*eve*		*false*	*false*	
$\ominus$	*tom*	*bob*	*zak*	*tom*	*ann*	*eve*		*false*	*false*	

Table 5.8: Propositional form of the *grandmother* learning problem.

DINUS – Back transformation to DDB clause form

The algorithm that converts the induced propositional concept definition to DDB clause form is fairly straightforward. First, transcribe the induced rules into DDB clauses (as in the constrained case). Than add the necessary determinate literals, ensuring that the values of the new variables can be uniquely determined from the values of old variables. For each clause repeat the following process, until all variables in the body of the clause and not in its head are introduced by determinate literals: choose a new variable and add to the clause the literal in L that introduced the new variable. The literals are then ordered according to the maximum depth of variables occurring in them.

To illustrate this transformation to a set of DDB clauses, the grandmother example is used. Suppose that a propositional learner induces the following two rules from the examples in Table 5.8:

$$\begin{array}{l} Class = \oplus \ \ \textbf{if} \ \ [A_1 = true] \\ Class = \oplus \ \ \textbf{if} \ \ [A_2 = true] \end{array}$$

This description is consistent with the examples, i.e., does not cover any negative example. (Note, however, that it is not too likely that any propositional learner would actually induce this description, since the first rule would suffice for discriminating between the positive and negative examples.) The two rules are transcribed to the clauses:

$$\begin{array}{lcl} grandmother(X, Y) & \leftarrow & mother(X, V). \\ grandmother(X, Y) & \leftarrow & mother(X, Z). \end{array}$$

The new variables V and Z in literals $mother(X, V)$ and $mother(X, Z)$ are not introduced by determinate literals, so the literals $father(V, Y)$ and $mother(Z, Y)$ from L have to be added to the first and second clause, respectively. The final form of the logic program would then be:

$$\begin{array}{lcl} grandmother(X, Y) & \leftarrow & father(V, Y), mother(X, V). \\ grandmother(X, Y) & \leftarrow & mother(Z, Y), mother(X, Z). \end{array}$$

Expressed in logic, this program stands for:

$$\begin{array}{ll} \forall X \forall Y : & grandmother(X, Y) \leftrightarrow \\ & [\exists V : f(V, Y) \wedge m(X, V)] \vee [\exists Z : m(Z, Y) \wedge m(X, Z)] \end{array}$$

or more precisely for:

$$\forall X \forall Y: \ grandmother(X,Y) \leftrightarrow$$
$$[\exists! V : f(V,Y) \wedge m(X,V)] \vee [\exists! Z : m(Z,Y) \wedge m(X,Z)]$$

as the value of each of the new variables is uniquely determined.

5.7.2 Learning recursive determinate DDB clauses

To learn recursive determinate definitions, Algorithm 5.8 needs to be slightly modified [Džeroski et al. 1992b]. Let us first distinguish between the case of recursive literals which do not and recursive literals which do introduce new variables. In the first case, membership queries are needed, and in the second, both membership and existential queries about the target relation are needed.

In the first case (when recursive literals do not introduce new variables), only steps (2) and (3) of Algorithm 5.8 need to be modified. In step (2), which constructs the list of literals F, we treat the target predicate p as any of the other background knowledge predicates and form all possible literals with it and the variables available, excluding the literal $p(X_1, X_2, ..., X_n)$. However, in step (3) we cannot evaluate features involving p by querying the background knowledge. In that case we first check whether the ground query is among the training examples for p, in which case we can determine its truth value (*true*, if the example is positive, *false* if negative). If, however, the query cannot be found among the examples, we have to resort to a membership query, that is, we must ask the user whether the answer to the query involved is true or false.

To illustrate the above modification to Algorithm 5.8, consider the task of learning the relation $member(X, Y)$, where X is of type *element* and Y is of type *list*. The background knowledge given includes the predicate $components(X, Y, Z)$, where X and Z are of type *list* and Y is of type *element*. This predicate is defined as $components([A|B], A, B)$ and decomposes a list $[A|B]$ into its head A and tail B. It is a flattened version of the *cons* function symbol. The equality predicate which works on arguments of the same type is also given. In this case j has to be at least 3. The maximum depth of variables is set to $i = 1$.

Table 5.9 gives four training examples and their transformation into propositional form. From the table we can see that the only determinate literal is $c(Y, A, B)$ (c and m stand for *components* and *member*, respectively), and $V_1 = \{X, Y, A, B\}$, where X and A are of type *element* and Y and B are of type *list*. Taking into account the types of arguments, the modified algorithm produces the list of features in the table. The literals $c(Y, X, Y)$, $c(Y, A, Y)$, $c(B, X, B)$, $c(B, A, B)$, $c(B, X, Y)$ and $c(B, A, Y)$ are also on the list of features, but have been excluded for the sake of readability (they always have the value *false*). Consider the propositional feature A_4 which corresponds to the recursive call $member(X, B)$. The value of this feature can be determined for the first example, since the ground query is in this case $member(1, [2])$, which can be found as a negative training example. However, for the other three examples, membership queries have to be posed (the answers are marked with an asterisk *). A similar observation holds for features A_5 and A_6.

	Vars		*New vars*		*Propositional features*					
$m(X,Y)$	X	Y	$c(Y,A,B)$		$c(Y,X,B)$	$X=A$	$Y=B$	$m(X,B)$	$m(A,Y)$	$m(A,B)$
Class	X	Y	A	B	A_1	A_2	A_3	A_4	A_5	A_6
$\oplus$	1	[1, 2]	1	[2]	*true*	*true*	*false*	*false*	*true*	*false*
$\oplus$	1	[2, 1]	2	[1]	*false*	*false*	*false*	$true^*$	$true^*$	$false^*$
$\oplus$	2	[2]	2	[]	*true*	*true*	*false*	$false^*$	*true*	$false^*$
$\ominus$	1	[2]	2	[]	*false*	*false*	*false*	$false^*$	*true*	$false^*$

Table 5.9: Propositional form of a recursive ILP problem.

A propositional learner might generate the following rules from the given propositional examples:

$$Class = \oplus \quad \textbf{if} \quad [A_2 = true]$$
$$Class = \oplus \quad \textbf{if} \quad [A_4 = true]$$

The rules would be transformed to the following definition:

$$member(X,Y) \ \leftarrow \ components(Y,A,B), \ =(X,A).$$
$$member(X,Y) \ \leftarrow \ components(Y,A,B), \ member(X,B).$$

which is the correct definition of the concept $member(X, Y)$.

The second case occurs if recursive literals with new variables are allowed, which are, for example, necessary for learning the *quicksort* program. In addition to the changes outlined above, the first substep of step (1) of Algorithm 5.8 has also to be changed. In this step, the target predicate p is treated exactly as the other predicates. Determining the values of the new variables in literals involving the target predicate requires in this case the use of *existential* queries about the target concept.

6

Experiments in learning relations with LINUS

This chapter discusses the performance of LINUS on four relation learning tasks taken from the machine learning literature. The domain descriptions are taken from Quinlan [1990] and the results of LINUS are compared to the results obtained by FOIL [Quinlan 1990]. An early version of FOIL (FOIL0) was used in the experiments. All the domains contain non-noisy data. The first three domains involve a very small number of examples. In the fourth domain, the training set of chess endgame positions is large enough to enable the comparison of the LINUS results with the results of FOIL and other machine learning systems in terms of classification accuracy.

6.1 Experimental setup

LINUS was used in the *relation learning mode* (see Section 5.4.4). In order to facilitate the comparison with FOIL, no literals of the form $X = a$ instantiating a variable to a constant were allowed in the induced predicate definitions. This was achieved by selecting only applications of utility predicates as new attributes to be used for learning (see Section 5.4.4).

The arguments of the background predicates used in FOIL were not typed and the same predicates could be used for different types of arguments. In LINUS, each such predicate was replaced by several predicates, one for each combination of predicate argument types. For example, the *adjacent* relation in the chess endgame example was replaced by two relations *adjacent_rank* and *adjacent_file* (see Section 6.5).

In the experiments, ASSISTANT and NEWGEM were used within

LINUS. The results obtained with ASSISTANT and NEWGEM on each of the domains were typically slightly different; they were all comparable to the results obtained by FOIL. In the chess endgame domain, the classification accuracy and the computational efficiency of LINUS was compared with the available results for FOIL, DUCE [Muggleton 1987] and CIGOL [Muggleton et al. 1989].

6.2 Learning family relationships

The family relationships learning task is described in [Hinton 1989] and [Quinlan 1989]. Given are two stylized families of twelve members each, as shown in Figure 6.1.

LINUS was used to learn the relation $mother(A, B)$ from examples of this relation and the background relations $father(X, Y)$, $wife(X, Y)$, $son(X, Y)$ and $daughter(X, Y)$. Negative examples were generated under the closed-world assumption (the *cwa* mode), both for LINUS and FOIL. To illustrate the post-processing feature of LINUS, we first present the intermediate clauses induced by LINUS using ASSISTANT (before post-processing).

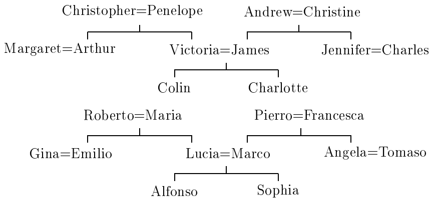

Figure 6.1: Two family trees, where = means *married to*, from Hinton [1989] and Quinlan [1989].

$$\begin{aligned} mother(A,B) &\leftarrow daughter(B,A), \\ &\qquad not\ father(A,B). \\ mother(A,B) &\leftarrow not\ daughter(B,A), \\ &\qquad son(B,A), \\ &\qquad not\ father(A,B). \end{aligned}$$

The second clause contains the irrelevant literal *not* $daughter(B, A)$, which was removed by post-processing. The resulting predicate definition is equal to the definition induced by LINUS using NEWGEM which contained no irrelevant literals. A comparison of the post-processed clauses with the clauses induced by FOIL is given below.

LINUS	FOIL
negative examples: *cwa*	negative examples: *cwa*
$mother(A,B) \leftarrow$ $daughter(B,A)$, *not* $father(A,B)$. $mother(A,B) \leftarrow$ $son(B,A)$, *not* $father(A,B)$.	$mother(A,B) \leftarrow$ $daughter(B,A)$, *not* $father(A,C)$. $mother(A,B) \leftarrow$ $son(B,A)$, *not* $father(A,C)$.

We can see that FOIL generated a more specific hypothesis, and that the definitions induced by LINUS and FOIL are very similar and both correct. The hypothesis induced by LINUS can be paraphrased as follows: 'A is the mother of B if B is a child of A and A is not the father of B.' The hypothesis induced by FOIL is more specific (if negation is interpreted as in Prolog), since a new existentially quantified variable appears in the negated literal. It can be summarized as: 'A is the mother of B if B is a child of A and A is not the father of anybody (C).' The definition contains a new variable C which stands for any person. While not necessary in this case, new variables are necessary when learning definitions of relations such as $grandmother(X, Y)$ in terms of the relations $mother(X, Y)$ and $father(X, Y)$. Thus, the hypothesis language of FOIL is more expressive than the one of LINUS which does not allow for the introduction of new variables.

6.3 Learning the concept of an arch

In this example, taken from Winston [1975] and Quinlan [1990], four objects are given. Two of them are arches (positive examples) and two are not (negative examples), as shown in Figure 6.2.

For inducing the target relation $arch(A, B, C)$, stating that A, B and C form an arch with columns B and C and lintel A, the following background relations were used: $supports(X, Y)$, $left\_of(X, Y)$, $touches(X, Y)$, $brick(X)$, $wedge(X)$ and $parallelepiped(X)$.

LINUS was first run with explicitly given (two) negative examples, and then with the negative examples generated in the *cwa* mode. The results of LINUS and FOIL are listed below.

LINUS	FOIL
negative examples: explicitly	negative examples: explicitly
$arch(A,B,C) \leftarrow$ $supports(B,A)$, *not* $touches(B,C)$.	no clauses, because of the encoding length restriction
negative examples: *cwa*	negative examples: *cwa*
$arch(A,B,C) \leftarrow$ $left\_of(B,C)$, $supports(B,A)$, *not* $touches(B,C)$.	$arch(A,B,C) \leftarrow$ $left\_of(B,C)$, $supports(B,A)$, *not* $touches(B,C)$.

The hypothesis induced with explicitly given negative examples can be paraphrased as follows: 'A, B and C form an arch if B supports A and B does not touch C'. Note that this definition is more general than the one obtained in the *cwa* mode; the latter has an additional literal stating that B must be left of C.

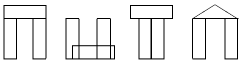

Figure 6.2: Arches and near misses, from Winston [1975] and Quinlan [1990].

With explicitly given negative examples, FOIL was unable to generate a hypothesis because of the encoding length restriction (described in Section 8.4) which restricts the total length of the hypothesis to the number of bits needed to enumerate the training examples explicitly. Using the *cwa* mode of negative examples generation, LINUS gave the same results as FOIL.

Analysis of the results

The hypothesis induced by both LINUS and FOIL covers also the case illustrated in Figure 6.3, which is a negative example of the arch concept. In the case when the negative examples are given explicitly this is reasonable, but it is counter-intuitive in the case where negative examples are generated under the closed-world assumption.

In order to get a more appropriate hypothesis, i.e., the same hypothesis as reported in Winston [1975], we have to be aware of the following problems:

- The problem of representation. Winston used a semantic net representation which is not equivalent to the relational representation selected in LINUS and FOIL. Some information was lost when transcribing the semantic net representation into the selected relations, e.g., the $part\_of$ relation which is essential in the semantic net representation was not included.
- Inappropriate use of the 'closed-world assumption' mechanism for generating negative examples.

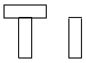

Figure 6.3: A covered negative example.

  - The set of positive examples is not exhaustive, therefore the set of negative examples includes also some positive examples of the arch concept.
  - Negative examples are generated in the 'object' space and not in the 'relation' space. Here the object space denotes the Cartesian product of the corresponding types of the n arguments of the target predicate (in the *cwa* mode all the possible combinations of values of n arguments of the target predicate are generated). In the relation space, each application of a utility predicate is added to the object space as an additional dimension, i.e., the relation space is the Cartesian product of k_{Attr} variables (see equation (5.4)). In this domain (as well as in the family relationships domain), relations are used to describe training examples and do not represent inherent domain knowledge (as is the case in the Eleusis example which follows). In the generation of negative examples, only the object space is affected; utility predicates are applied to the generated negative examples. Since the relations are essential to describing training examples and are not affected, not all possible negative examples are generated.

- Insufficient set of training examples. The hypotheses induced by LINUS and FOIL are correct with regard to the given training set. However, they cover the negative example given in Figure 6.3. As will be shown below, the obtained results can be improved by selecting a more appropriate set of training examples.

Learning from an enlarged training set

LINUS was used on an enlarged set of training examples, given in Figure 6.4, which includes two additional negative examples.

From the enlarged training set, containing two additional negative examples, the hypothesis induced by LINUS using both ASSISTANT and NEWGEM is essentially the same as the one reported by Winston [1975]:

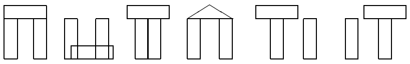

Figure 6.4: An enlarged training set of arches and near misses.

$$\begin{array}{lll} arch(A,B,C) & \leftarrow & supports(B,A), \\ & & supports(C,A), \\ & & not\ touches(B,C). \end{array}$$

The induced hypothesis corresponds to the part of the semantic net description consisting of 'must' links only [Winston 1975].

6.4 Learning rules that govern card sequences

The Eleusis learning problem was originally addressed by the SPARC/E system [Dietterich and Michalski 1986]. The description here is taken from Quinlan [1990]. In the Eleusis card game, the dealer invents a secret rule specifying a condition under which a card can be added to a sequence of cards. The players attempt to add a card to the current sequence. If a card is a legal successor it is placed to the right of the last card, otherwise it is placed under the last card. The horizontal *main line* represents the sequence as developed so far, while the vertical *side lines* show incorrect plays. Three layouts, reproduced from Quinlan [1990], are given in Figure 6.5.

Each card other than the first in the sequence provides an example for learning the target relation *can_follow*. The example is labeled ⊕ if the card appears in the main line, and ⊖ if it is in a side line. The target relation $can\_follow(A,B,C,D,E,F)$ states that a card of rank A and suit B can follow a sequence ending with: a card of rank C and suit D; E consecutive cards of suit D; and F consecutive cards of the same color. Background relations that can be used in induced clauses are the following: $precedes\_rank(X,Y)$, $precedes\_suit(X,Y)$, $lower\_rank(X,Y)$, $face(X)$, $same\_color(X,Y)$, $odd\_rank(X)$, and $odd\_num(X)$.

main line	A♡	7♣	6♣	9♠	10♡	7♡	10♢	J♣	A♢	4♡	8♢	7♣	9♠
side lines			K♢			5♠		Q♢		3♠			9♡
			J♡										6♡
main (ctd)	10♣	K♠	2♣	10♠	J♠								
side (ctd)		Q♡											
		A♢											
main line	J♣	4♣	Q♡	3♠	Q♢	9♡	Q♣	7♡	Q♢	9♢	Q♣	3♡	K♡
side lines	K♣	5♠				4♠		10♢					
		7♠											
main (ctd)	4♣	K♢	6♣	J♢	8♢	J♡	7♣	J♢	7♡	J♡	6♡	K♢	
mainline	4♡	5♢	8♣	J♠	2♣	5♠	A♣	5♠	10♡				
side line	7♣	6♠	K♣	A♡		6♣		A♠					
	J♡	7♡	3♡	K♢									
	4♣	2♣		Q♠									
	10♠	7♠											
	8♡	6♢											
	A♢	6♡											
	2♢	4♣											

Figure 6.5: Three layouts of the Eleusis card game, from Dietterich and Michalski [1986] and Quinlan [1990].

In the first layout, the intended dealer's rule was: 'Completed color sequences must be of odd length and a male card may not appear next to a female card'. Neither FOIL nor LINUS could discover the intended rule, because no information on the gender of cards was encoded in the background relations. LINUS using ASSISTANT with post-processing induced the clauses given below.

$$\begin{array}{lll} can\_follow(A,B,C,D,E,F) & \leftarrow & same\_color(B,D). \\ can\_follow(A,B,C,D,E,F) & \leftarrow & odd\_num(F), \\ & & not\ precedes\_suit(D,B). \\ can\_follow(A,B,C,D,E,F) & \leftarrow & odd\_num(F), \\ & & not\ precedes\_rank(C,A). \end{array}$$

While the clauses induced by LINUS using both ASSISTANT and NEWGEM cover all the positive examples, the ones induced by FOIL

are not complete: they do not cover the positive example $can\_follow(10, hearts, 9, spades, 1, 3)$. This is due to the encoding length stopping criterion which prevents further search for clauses. Although the encoding length restriction is useful when dealing with noisy data, it is unsuitable for non-noisy domains. The use of the encoding length restriction could be controlled by a parameter: it should be applied if the domain is noisy or inexact, but should not be applied to exact domains with no noise, such as the domains described in this chapter. Here are the clauses induced by LINUS using NEWGEM and FOIL:

LINUS using NEWGEM	FOIL
layout 1	layout 1
negative examples: explicitly	negative examples: explicitly
$can\_follow(A,B,C,D,E,F) \leftarrow$ $same\_color(B,D).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $same\_color(B,D).$
$can\_follow(A,B,C,D,E,F) \leftarrow$ $odd\_num(F),$ $odd\_rank(A).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $odd\_num(F),$ $odd\_rank(A).$
$can\_follow(A,B,C,D,E,F) \leftarrow$ $notface(A),$ $lower\_rank(C,A).$	

In the second layout, both LINUS using NEWGEM and FOIL correctly induced the intended rule: 'Play alternate face and non-face cards'.

LINUS using NEWGEM	FOIL
layout 2	layout 2
$can\_follow(A,B,C,D,E,F) \leftarrow$ $face(A),$ $notface(C).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $face(A),$ $notface(C).$
$can\_follow(A,B,C,D,E,F) \leftarrow$ $face(C),$ $notface(A).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $face(C),$ $notface(A).$

LINUS using ASSISTANT (with post-processing) generated a hypothesis containing a superfluous clause which was not removed in post-processing.

$$\begin{aligned} can\_follow(A,B,C,D,E,F) &\leftarrow face(C), \\ &\quad\; not\ face(A). \\ can\_follow(A,B,C,D,E,F) &\leftarrow face(A), \\ &\quad\; not\ face(C). \\ can\_follow(A,B,C,D,E,F) &\leftarrow not\ odd\_num(E). \end{aligned}$$

In the third layout, the intended rule is: 'Play a higher card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card'. FOIL discovered only one clause, approximately describing the first part of the rule. LINUS using ASSISTANT discovered an approximation of the whole rule: 'Play a higher or equal card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card'. The induced clauses are given below.

$$\begin{aligned} can\_follow(A,B,C,D,E,F) &\leftarrow lower\_rank(A,C), \\ &\quad\; precedes\_suit(D,B). \\ can\_follow(A,B,C,D,E,F) &\leftarrow precedes\_suit(B,D), \\ &\quad\; not\ lower\_rank(A,C). \end{aligned}$$

Again, the hypothesis induced by LINUS is complete, while FOIL's is not. Using NEWGEM, LINUS generated exactly the intended dealer's rule.

LINUS using NEWGEM layout 3	FOIL layout 3
$can\_follow(A,B,C,D,E,F) \leftarrow$ $lower\_rank(A,C),$ $precedes\_suit(D,B).$	
$can\_follow(A,B,C,D,E,F) \leftarrow$ $lower\_rank(C,A),$ $precedes\_suit(B,D).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $not\ lower\_rank(A,C),$ $precedes\_suit(B,D).$