FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A workbench to develop ILP systems

João Azevedo

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Camacho (PhD)

July 2010

A workbench to develop ILP systems

João Azevedo

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Eduarda Rodrigues (Professor Auxiliar Convidado)
External Examiner: Nuno Lau (Professor Auxiliar)
Supervisor: Rui Camacho (Professor Associado)

22nd July, 2010

Abstract

Inductive Logic Programming (ILP) is a sub-field of Machine Learning that provides an excellent framework for Multi-Relational Data Mining applications. ILP has been used in both industrial and scientific complex and relevant problems. ILP aims at a formal framework as well as practical algorithms for inductive learning of relational descriptions in the form of logic programs. ILP inherits the sound theoretical basis from Logic Programming and the experimental approach and operation towards practical algorithms from Machine Learning. ILP is an exciting field of research, still showing a good margin for progress.

There is a wide range of lines of research to overcome current shortcomings of existing ILP systems. It is common practise that when researching on a new technique the researcher has to develop his own system or spend a considerable amount of time studying the implementation details of an existing system in order to evaluate his new techniques. Since the initial conceptual proposal of Inductive Logic Programming many ILP systems have been developed. Around 100 ILP systems have been developed to date. Thus, it is natural to find the many techniques that have been proposed to improve the efficiency of ILP systems scattered among many systems.

Having an integrated framework, containing the most interesting techniques of ILP in a modular architecture should be interesting for the progress on the area, by providing practitioners and curious users with an easier way to experiment on the area and to test new techniques in a relatively straightforward way.

The work reported in this document aims to propose a tool to include all major relevant techniques for the development of ILP systems. The tool is based on a modular architecture of the system and permits the assemblage of new ILP systems by just combining modules. By choosing different sets of modules the user may construct different types of ILP systems. We have also developed and implemented in the tool a new parallel algorithm. A prototype of the tool is the material outcome of our work.

Resumo

A Indução de Programas em Lógica (*Inductive Logic Programming* - ILP) é uma sub-área da Aprendizagem Computacional que proporciona uma excelente *framework* para aplicações de extracção de conhecimento multi-relacional. ILP tem vindo a ser usado em problemas complexos e relevantes, tanto na indústria como no meio científico. O ILP procura fornecer tanto uma *framework* formal como algoritmos práticas para a aprendizagem, através do mecanismo de indução, de descripções relacionais sob a forma de programas em lógica. ILP herda a base teórica da programação em lógica e a aproximação experimental da aprendizagem computacional. O ILP é, neste momento, uma área de investigação com uma enorme margem para progresso.

Existe actualmente uma vasta gama de investigação para colmatar alguns dos problemas dos sistemas de ILP actuais. É prática comum que, quando se investiga um nova técnica, o investigador tenha de desenvolver o seu próprio sistema, ou dispender uma quantidade considerável de tempo estudando os detalhes de implementação de um sistema existente como forma de avaliar o seu etudo. Desde a primeira proposta conceptual da Indução de Programas em Lógica, vários sistemas de ILP foram desenvolvidos. Estima-se que cerca de 100 devam existir, à data. Assim, é natural encontrar as várias técnicas que vieram sendo propostas para melhorias da eficiência de sistemas de ILP espalhadas pelos vários sistemas.

Ter acesso a uma *framework* integrada, contendo as técnicas mais interessantes de ILP numa arquitectura modulas deve ser interessante para o progresso na área, fornecendo aos praticantes ou utilizadores curiosos uma maneira mais fácil de experimentar na área e de testar novas técnicas de uma forma relativamente rápida.

Este documento procura propor uma ferramenta que inclua todas as principais técnicas de ILP para o desenvolvimento de sistemas. Começa por introduzir a área da Indução de Programas em Lógica, seguindo-se a descrição de alguns dos sistemas mais relevantes até à data. Depois, é dada uma descrição da arquitectura modular do sistema, bem como dos mecanismos de configuração da ferramenta. Finalmente, algumas conclusões e trabalho futuro na ferramenta são apresentados.

Acknowledgements

I would like to thank to everyone who somehow contributed for the successful work on this thesis.

I would like to thank my supervisor, Prof. Rui Camacho, for the constant help and support during the development of this thesis. His valuable insights and feedback were fundamental for this work and for the understanding of many ILP techniques.

To my friends from FEUP, for their companionship, strength, support and all the great moments over the past five years.

To my family, for their constant support, understanding and inspiration.

Copyright Acknowledgements

The examples given for the description of inductive concept learning in section 2.2.1, and the daughter example in 2.2 were taken from [LD94]. Figures 2.1, 2.2, 2.3 and 2.4 were taken from [LD94]. Figures 4.2 and 4.3 were taken from [FCR$^+$09]. Some of the descriptions of systems in chapter 3 were taken from [LWZ$^+$96, Dze].

Contents

List of Figures

LIST OF FIGURES

List of Tables

List of Algorithms

Abreviations and Symbols

AG	Accuracy Gain
API	Application Programming Interface
BF	Branching Factor
CWA	Closed World Assumption
DNA	Deoxyribonucleic Acid
EBL	Explanation-Based Learning
ID3	Iterative Dichotomiser 3
IG	Information Gain
ILP	Inductive Logic Programming
IR	Inverse Resolution
LGG	Least General Generalization
LOC	Length of Clause
LP	Logic Programming
MDIE	Mode Directed Inverse Entailment
MIS	Maximal Independent Set
MIS	Model Inference System
ML	Machine Learning
NEC	Node Evaluation Cost
RL	Range List
SLD	Selective Linear Definite
WAG	Weighted Accuracy Gain
WAM	Warren Abstract Machine
WIG	Weighted Information Gain
WWW	World Wide Web
XML	eXtended Markup Language

Chapter 1

Introduction

ILP is a research area at the intersection of Machine Learning and Logic Programming. The term was first coined by Stephen Muggleton, in 1991 [Mug91]. ILP aims at a formal framework and practical algorithms for inductive learning of relational descriptions in the form of logic programs [LD94]. ILP inherits the sound theoretical basis from Logic Programming [Bra00] and the experimental approach and orientation towards practical applications from Machine Learning [Mit97].

ILP systems have been successfully applied to a wide range of domains that include Molecular Biology [MKS92], Biochemistry [KMLS92] and environmental monitoring [DDRW95]. Generally, they have been used in classification and regression tasks as well as clustering and finding Association Rules. The richness of ILP representation language make these algorithms adequate for (Multi-)Relational Data Mining tasks. Despite their considerable successes, ILP systems suffer from significant limitations that reduce their large scale applicability. Most ILP systems execute in main memory, limiting their ability to process large databases. ILP systems are computationally expensive, eg. evaluating individual rules may take considerable time, even for small data sets [BGSS99]. Search procedures used by ILP systems are highly redundant. ILP systems have poor human-computer interaction. In summary there are a large number of research lines to overcome current shortcomings of existing systems. It is common practise that when researching on a new technique the researcher has to develop his own system or spend a considerable amount of time studying the implementation details of an existing system in order to evaluate his new techniques. Since the initial conceptual proposal of Inductive Logic Programming many ILP systems have been developed. In a presentation at the ILP 2005 conference, Ashwin Srinivasan pointed out that around 100 ILP systems have been developed to date. Thus, it is natural to find the many techniques that have been proposed to improve the efficiency of ILP systems scattered among many systems. Understanding which techniques, or combination of techniques, contribute the most is therefore a rather difficult task because one usually compares the techniques in different contexts (eg. different ILP systems or programming languages). Moreover, the effects on performance of combining several techniques or the impact of a new technique when used in conjunction with existing techniques is often not studied. Apart from efficiency, the following five areas have been, or are being, studied: Incorporating probabilities. Novel search methods. Techniques for parallel ILP. Using special-purpose reasoners. Enhancing human-computer interaction.

1.1 Problem

Once again, the research results are scattered by a large number of ILP systems. It would be of significant value to have a tool that incorporates many ILP advances within a single system. This need is well expressed in Srinivasans words at the ILP 2005 invited talk: :. It is my belief - unsurprising, given my own training and inclination - that we need a well-defined engineering R&D project to implement and apply conceptual advances being made in these five areas. Consider what this means. It means a band of engineers working together to design and develop an ILP system that is routinely updated to incorporate advances made in each of the five areas. As with any well-engineered tool, demonstrations of robustness and efficiency using standard tests will be necessary. It is also common for such projects to be driven by a performance wish-list... Ashwin Srinivasan, IBM India Research Laboratory, India Five Problems in Five Areas for Five Years invited talk at ILP 2005 conference (Bon, Germany)

1.2 Motivation and Objectives

This report thus presents an architecture for a worbench enabling the simple creation of ILP systems, called BET workbench. This workbench incorporates most of the basic functionalities required by a wide range of ILP systems and provides a set of libraries of predicates, ready to use and providing valuable basic functionalities such as numerical libraries, basic graphical interfaces and basic primitives for parallel execution. The defined architecture should be module-based and define different abstraction layers, corresponding to different levels of features for an ILP system. By setting up incompatibilities and dependencies, it should be possible to easily create ready to use systems.

1.3 Document Structure

This document is organized in 6 different chapters. Chapter 1, the current chapter, describes the topics presented here and the overall document structure. Chapter 2 introduces the field of ILP. This introduction starts by describing some background history and the reasons for the appearence of this research area. To better understand some of its theoretical basis, some terminology is introduced and described. An overview of the most common techniques on ILP systems is then presented. To end the chapter, some real-world applications where the results of applying ILP techniques were highly successful are shown. Chapter 3 serves as a description of some of the most famous currently available ILP systems. The choice for particularly describing these, and not other systems is based on the contribution of those systems for the research field, namely by introducing new techniques able to be incorporated on the development tool which is the subject of thesis work. Chapter 4 describes the modules contained in the BET workbench, its architecture and the way it can be configurated. Chapter 5 introduces some parallelization techniques and describes an parallel algorithm for MDIE-based systems, implemented as a module for the BET workbench. Chapter 6 draws some conclusions on the topic and describes the future work on the workbench.

Chapter 2

Introduction to ILP

Inductive Logic Programming is a research area at the intersection of Machine Learning and Logic Programming. It aims to provide both a formal framework and practical algorithms for inductively learning logic programs from examples. This chapter gives an introduction to ILP on both perspectives. In order to better understand the movitations behind its appearance, a short background on deductive and inductive logic is presented. After that, some basic concepts to understand ILP are introduced, followed by a description of the currently used techniques. Some advanced topics on the area are then introduced. To better understand the capabilities of this research area, some successful applications of ILP are described.

2.1 Background

The first formal study of logic appeared in the Prior Analytics [Ari], Aristotle's work on deductive reasoning, specifically the syllogism. Despite that, only during the 19th and 20th centuries theoreticians such as Boole and Frege transformed it into a rigorous mathematical science. Since then, a school of philosophers have been promoting the view that logic is the ultimate foundation of all sciences, not only mathematics. These philosophers are known as logical positivists. This view assumes that the logical language of first order predicate calculus is able to phrase every mathematical statement, and all the valid scientific reasoning is based on the logical derivation of axioms. Gödel's demonstration [Gö01] that a small collection of sound rules of inference was complete for deriving all consequences of formulae in first order predicate calculus served as a powerful argument for logical positivism. Later, Robinson demonstrated [Rob65] that a single rule of inference, called resolution, is both sound and complete for proving statements within this calculus. In order to apply resolution, formulae are normalised to what is known as *clausal form*.

Robinson's discovery lead to huge progresses [Kow79] on the application of logic within computer science, ultimately leading to the development of the logic based programming language Prolog [1] and automatic deduction systems . Prolog statements are phrased in a restricted clausal form called Horn clause logic, and computations take the form of logical proofs based on the application of the rule of resolution. Prolog has naturely developed into a widely used programming language [Bra00, Ste94] and spawned the rigorous theoretical school of Logic Programming [Llo84].

A certain question has cropped up throughout the development of logical deduction: if all human and computer reasoning proceeds from logical axioms, then where do logical axioms proceed from? The widely accepted answer to this claims that logical axioms can be constructed from particular facts using *inductive* reasoning, therefore leading to generalised beliefs. The history of inductive reasoning interacts with the development of its deductive counterpart. In fact, inductive reasoning played a key role in Socrates' dialectic discussions in ancient Greece [Pla08]. In these discussions, concepts were developed and refined using a series of examples and counter-examples drawn from every day life. The first detailed description of the inductive scientific method was introduced in the *Novum Organum* [Bac09], by Francis Bacon. Statistics, a mathematical discipline developed from methods for predicting the outcome of games of chance, went on to have a central role in the evaluation of scientific hypothesis. Gödel's incompleteness theorem [Gö01] prompted Turing to attempt to show [Tur39] that problems concerning incompleteness of logical theories could be overcome by the use of an *oracle* capable of verifying underivable statements. Later work by Turing showed that Gödel's incompleteness theorem required that intelligent machines be capable of learning from examples [Tur47]. Various logical positivists have developed statistical theories for confirming scientific hypothesis posed in first order logic. Although there were developments on computer-based inductive systems within the framework of full first order logic, namely by Plotkin [Plo71] and Shapiro [Sha05], most successes within the field of Machine Learning have derived from systems which construct hypotheses within the limits of propositional logic. The major successes here have been in the area of inductive construction of expert systems [Mug90]. From a software engineering perspective, some of the inductively constructed expert systems show sizeable reductions in development and maintenance times, when comparing to the ones developed in a traditional fashion. Such an observation supports the idea that inductive inference should play an increasingly important role in future software development.

Despite being successful, the described inductive systems show some limitations. The representation of data is one of the limitations, as propositional level systems cannot be used in areas requiring essentially relational knowledge representations. Problems also occur in areas involving arbitrarily complex structural relationships, such as prediction

[1] Acronym of Programming in Logic

of protein folding and DNA gene mapping. Inductive algorithms such as ID3 [Qui] use only a fixed set of attributes attached to each example, showing an inability to make use of background knowledge. Explanation-based learning [MKKC86] attempted to overcome this limitation by redefining the learning problem, constraining hypothesis to those being derivable from background knowledge. However, since background knowledge is rarely complete in applications, the constraint posed by EBL is now generally believed to be over-restrictive. Another limitation of the described inductive systems is related to a strong bias of vocabulary, as the hypotheses are constructed within the limits of a fixed vocabulary of propositional attributes.

2.2 Basic concepts

This section introduces some concepts necessary for a complete understanding of ILP. The reader is first presented with the basics of inductive concept learning, followed by a description of the roles of background knowledge and language bias. The problem of handling imperfect data is then posed. The concepts introduced here form the theoretical basis of ILP, thus are needed for a complete understanding of the practical algorithms presented in the following sections.

2.2.1 Inductive concept learning

The goal of Machine Learning, as part of computer science, is to develop methods, techniques and tools for building intelligent learning machines. The concept of intelligent learning machines is described as learning programs which are able to change themselves in order to perform better at a given bask. To perform better means, for example, to perform more efficiently or more accurately. Better might also be related to the learner's hability to handle a broader class of problems.

Machine Learning paradigms [Car90] include inductive learning, deductive learning, learning with generic algorithms and connectionist learning (learning with neural networks). It is possible to integrate multiple learning paradigms in a single multistrategy learning system [MT91]. Mitchie's strong criterion [Mic88] 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.

Inductive logic programming is a subfield of inductive learning. It is 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 constructed. The problems addressed by this 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. Considering $\mathcal{U}$ a universal set of objects (or observations), a concept $\mathcal{C}$ can be formalized as a subset of objects in $\mathcal{U} : \mathcal{C} \subseteq \mathcal{U}$. Therefore, the task of learning a concept $\mathcal{C}$ can be formally described as the task of learning to recognize objects in $\mathcal{C}$, i.e., to be able to tell whether $x \in \mathcal{C}$ for each $x \in \mathcal{U}$.

In Machine Learning, a formal language for describing objects and concepts needs to be selected. Objects are usually 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 an enumeration of their features, called attributes, each of them taking a value from a corresponding pre-defined value set. For instance, if we want to describe playing card games we can use two attributes: suit and rank. The set of values for the attribute *Suit* is {*hearts, spades, clubs, diamonds*} and the set of values for *Rank* is {*2, 3, 4, 5, 6, 7, 8, 9, 10, j, q, k, a*}, where *j, q, k, a* stand for *jack, queen, king* and *ace*, respectively. With the previously described attributes, 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, the same object can be described by the tuple $< diamonds, 7 >$. In the first-order language of Horn clauses, used in logic programming, cards are described by ground facts, e.g., $card(diamonds, 7)$. In this case, *card* is a predicate symbol, and the values of the two arguments *Rank* and *Suit* of the predicate $card/2$ are the constants *diamonds* and 7.

In an object description language, a pair of cards can be described by comprising four attributes: the suit and rank of each 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 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 possible descriptions are $pair(diamonds, 7, hearts, 7)$, and the more expressive $pair(card(diamonds, 7), card(hearts, 7))$, where *pair* is a predicate symbol and *card* is a function symbol. Generalizing from the previous description, one can define the concept *pair*. Concepts can be described extensionally or intensionally. A concept is described extensionally by listing the descriptions of all of its instances. Such a representation is usually undesirable, namely because a concept may contain an infinite number of instances. Consequently, it is preferable to describe concepts intensionally. Such a description requires the usage of a concept description language, allowing for more compact and concise concept description, e.g., in the form of rules. We can, for instance, describe the concept *pair* by the following rule:

$$\begin{aligned} \textit{pair} \quad \textbf{if} \quad & [\textit{Rank}_1 = 2] \wedge [\textit{Rank}_2 = 2] \vee \\ & [\textit{Rank}_1 = 3] \wedge [\textit{Rank}_2 = 3] \vee \\ & \ldots \\ & [\textit{Rank}_1 = a] \wedge [\textit{Rank}_2 = a] \end{aligned}$$

Since the attributes $\textit{Suit}_1$ and $\textit{Suit}_2$ do not appear in the rule, they can have any value in the appropriate range. If the concept language supports the use of variables, the *pair* concept can be described in a much more compact way:

$$\textit{pair} \quad \textbf{if} \quad \textit{Rank}_1 = \textit{Rank}_2$$

In the language of Horn clauses, the same concept description assumes the following form:

$$\textit{pair}(\textit{Suit}_1, \textit{Rank}_1, \textit{Suit}_2, \textit{Rank}_2) \leftarrow \textit{Rank}_1 = \textit{Rank}_2$$

The same description can take a more expressive form, in case the description language allows for the use of function symbols:

$$\textit{pair}(\textit{card}(\textit{Suit}_1, \textit{Rank}_1), \textit{card}(\textit{Suit}_2, \textit{Rank}_2)) \leftarrow \textit{Rank}_1 = \textit{Rank}_2$$

Having a symbolic description language for both objects and concepts, a procedure is needed that will establish whether a given object belongs to a given concept. Assuming an object and concept description language as stated above, one can say that an object belongs to a concept if the object description satisfies the concept description. Whenever that happens, the concept description is said to cover the object description. The following examples assume both an object and concept description based on Horn clauses. Taking that into account, an object description should, from now on, be called a fact, and the intensional concept description to be learned should be called an hypothesis. In order to learn a concept $\mathcal{C}$ we need to generalize from a set of examples $\mathcal{E}$. Such a set will contain facts, labeled $\oplus$ or $\ominus$ in case the object is an instance of the concept $\mathcal{C}$ or otherwise, respectively. The facts from $\mathcal{E}$ labeled $\oplus$ form the set of positive examples ($\mathcal{E}^+$) and those labeled $\ominus$ form the set of negative examples ($\mathcal{E}^-$).

In single concept learning, a concept description is induced from facts labeled $\oplus$ and $\ominus$. $\oplus$ labeled facts are called examples and $\ominus$ labeled facts are called counter-examples 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 can be divided into subsets of examples for each individual concept. The problem of single concept learning can now be formally stated as, given a set $\mathcal{E}$ of positive and negative examples of concept $\mathcal{C}$, find a hypothesis $\mathcal{H}$ 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

$$covers(\mathcal{H}, e) \tag{2.1}$$

can be introduced, which returns the value *true* if e is covered by $\mathcal{H}$, and *false* otherwise. In Logic Programming, where a hypothesis is a set of program clauses and an example is a ground fact, a forward chaining procedure [Bry90] or a form of SLD-resolution rule of inference [Llo84] can be used to check whether e is entailed by $\mathcal{H}$. The function $covers(\mathcal{H}, e)$ can be extended to work for sets of examples in the following way:

$$covers(\mathcal{H}, \mathcal{E}) = \{e \in \mathcal{E} | covers(\mathcal{H}, e) = true\} \tag{2.2}$$

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. A hypothesis $\mathcal{H}$ is complete with respect to a set of examples if it covers all the positive ones. A hypothesis $\mathcal{H}$ is consistent with respect to a set of examples if it covers none of the negative examples. There is an alternative to the completeness and consistency criteria, known as the *quality criterion* [Rae92].

One of the major aims of learning is to classify unseen objects with respect to $\mathcal{C}$. Therefore, the induced hypothesis $\mathcal{H}$ can also be viewed as a classifier of new objects. This poses a new criteria of sucess for the learning systems, as the previous definition of inductive concept learning required that both $\mathcal{C}$ and $\mathcal{H}$ agreed on all examples from $\mathcal{E}$. From this perspective, the accuracy of classifying unseen objects is the main criterion of sucess in the learning system. The accuracy measures the percentage of objects correctly classified by the hypothesis. Other important performance criteria of learning include the extent to which the hypothesis is understood by humans, its statistical significance and the information content.

2.2.2 Background knowledge

When building a model, if no prior knowledge about the learning problem is given, it learns exclusively from examples. However, there is usually a significant number of difficult problems requiring a substantial body of prior knowledge. This kind of declarative prior knowledge is known as *background knowledge*. Since concept learning can be viewed as searching the space of concept descriptions [Mit82], the hypothesis language $\mathcal{L}$ and the background knowledge $\mathcal{B}$ determine the search space of possible concept descriptions, also known as the *hypothesis space*. The inclusion of background knowledge in the problem statement of inductive concept learning requires that the hypothesis $\mathcal{H}$ to be found be complete and consistent with respect to both the examples $\mathcal{E}$ and background knowledge $\mathcal{B}$.

This formulation requires a revision of the function *covers* introduced in equations 2.1 and 2.2, in order to take background knowledge $\mathcal{B}$ into account:

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

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

The completeness and consistency requirements are also modified, such that $covers(\mathcal{B},\mathcal{H},\mathcal{E}^+) = \mathcal{E}^+$ (for completeness) and $covers(\mathcal{B},\mathcal{H},\mathcal{E}^-) = \emptyset$ (for consistency) are verified.

2.2.3 Language bias

A bias is a mechanism employed by a learning system to constrain the search for hypothesis [UM82]. Bias can either determine how the hypothesis space is searched (*search bias*) or determine the hypothesis space itself (*language bias*).

By selecting a stronger language bias (a less expressive hypothesis language) the search space becomes smaller and learning more efficient. However, one must take into account the expressiveness/tractability tradeoff, as this may prevent the system from finding a solution which is not contained in the less expressive language.

Different logical formalisms have been used in inductive learning systems to represent examples and concept descriptions. Within this scope, one most frequently distinguishes between systems which learn attribute descriptions, and systems which learn first-order relational descriptions. Inductive learning algorithms, such as ID3 [Qui86] and AQ [Mic83] use an attribute-value language to represent objects and concepts, therefore called *attribute-value learners* or *propositional learners*. The 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 are called *relational learners*, due to their ability to induce descriptions of relations (definitions of predicates). Since, in these systems, objects can be described structurally, i.e., in terms of their components and relations between them, background knowledge can easily be integrated. In relational learners, the languages used to represent examples, concepts and background knowledge are typically subsets of first-order logic. The language of *logic programs* [Llo84] provides sufficient expressiveness for solving a significant number of relational learning problems. Learners that induce hypotheses in the form of logic programs are therefore called *inductive logic programming* systems.

2.2.4 Imperfect data

The definition of the inductive concept learning task introduced in section 2.2.1 required that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. 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. A 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.

The following kinds of data imperfection are usually found when learning relations from real-world data:

- 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;
- missing values in the training examples.

Learning systems usually have a mechanism for dealing with the first three types of imperfect data alltogether, often called *noise-handling mechanisms*. These kind of systems are typically designed to prevent the induced hypothesis from *overfitting* the training set of examples. Overfitting refers to a situation where the concept description $\mathcal{H}$ agrees with $\mathcal{C}$ perfectly on the training examples, but poorly on unseen examples. That is, $\mathcal{H}$ captures some regularities that are not genuine. In order to avoid this behaviour, there is the need to relax the completeness and consistency criteria in the definition of the inductive learning task and replace them by a more general quality criterion allowing the hypothesis to misclassify some of the training examples.

2.3 Inductive Logic Programming

A learning system whose hypothesis language ($\mathcal{L}$) is the language of logic programs is basically synthesizing a logic program. Relational learners who produce this kind of hypotheses are known to apply *Inductive Logic Programming*(ILP) [Mug91, Mug92]. In ILP systems, the training examples, the background knowledge and the induced hypothesis are all expressed in a logic program form. Some restrictions are imposed on each of the three languages, in order to reduce the search space. Training examples, for instance, are typically represented as ground facts of the target predicate.

In order to illustrate the ILP task, it is convenient to introduce some simple problems, based on family relations.

Example 2.1 (An ILP problem: defining the *sibling* relationship)**.** The learning task is to define the target relation $sibling(X,Y)$, meaning that person X is a sibling of person Y, in terms of background knowledge relation $parent$. For the sake of simplicity, the $parent$ relation won't be defined in terms of relations $mother$ and $father$, having no sexual distinctions of persons. The relations used in the background knowledge are given in Table 2.1.

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

$$sibling(X,Y) \leftarrow parent(X,Z), parent(Y,Z)$$

This definition states that a sibling of someone is someone who shares a parent with him/her. This definition is consistent and complete with respect to the background knowledge and the training examples.

Training examples		*Background knowledge*
$sibling(bob,alice).$	$\oplus$	$parent(bob,george).$
$sibling(alice,bob).$	$\oplus$	$parent(alice,george).$
$sibling(harriet,george).$	$\ominus$	$parent(bob,harriet).$
$sibling(harriet,bob).$	$\ominus$	$parent(alice,harriet).$
$sibling(harriet,alice).$	$\ominus$	
$sibling(bob,harriet).$	$\ominus$	
$sibling(bob,george).$	$\ominus$	
$sibling(george,harriet).$	$\ominus$	
$sibling(george,bob).$	$\ominus$	
$sibling(george,alice).$	$\ominus$	
$sibling(alice,harriet).$	$\ominus$	
$sibling(alice,george).$	$\ominus$	

Table 2.1: A simple ILP problem: learning the *sibling* relationship

Example 2.2 (An ILP problem: defining the *daughter* relationship)**.** The task of this example, taken from [LD94] is to define the target $daughter(X,Y)$, which states that person X is a daughter of person Y, in terms of background knowledge $female$ and $parent$. These relations are given in Table 2.2. There are two positive and two negative examples of target relation.

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

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

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

Table 2.2: A simple ILP problem: learning the *daughter* relationship

This definition is consistent and complete with respect to the background knowledge and the training examples.

Systems capable of solving the kind of learning tasks as the one described above can be divided along several dimensions. They can learn either a single predicate or multiple predicates simultaneously. 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). During the learning process, the system may rely on an oracle to verify the validity of generalizations or to classifiy examples generated by the learner. The learner is called interactive in this case and non-interactive otherwise. A learner may try to learn a concept from scratch or accept an initial hypothesis which is revised in the learning process. The latter are called theory revisors.

ILP systems are usually divided into ones which are batch non-interactive systems that learn a single predicate from scratch (*empirical ILP systems*) and the ones which are interactive and incremental theory revisors that learn multiple predicates (*interactive ILP systems*) [Rae92].

To make the definition of learning presented in section 2.2.1 operational for ILP, the notion of coverage must be introduced and used in the definitions of the function *covers* in equations 2.3 and 2.4:

$$covers(\mathcal{B},\mathcal{H},e) = true \ \ if \ \ \mathcal{B} \cup \mathcal{H} \models e \tag{2.5}$$

$$covers(\mathcal{B},\mathcal{H},\mathcal{E}) = \{e \in \mathcal{E} | \mathcal{B} \cup \mathcal{H} \models e\} \tag{2.6}$$

The symbol $\models$ stands for logical entailment [Llo84], and, in the previous equations, has the meaning that e is a logical consequence of $\mathcal{B} \cup \mathcal{H}$. To see if an example is logically entailed by $\mathcal{B} \cup \mathcal{H}$, usually the SLD-resolution proof procedure is used [Llo84]. An alternative is to use the forward chaining procedure [Bry90]. A more detailed discussion of the proof procedures in ILP systems is available in the book *Interactive Theory Revision: An Inductive Logic Programmming Approach* [Rae92].

2.3.1 Context

The usual context for ILP is as follows. The learning agent is provided with background knowledge $\mathcal{B}$, positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$ and constructs an hypothesis $\mathcal{H}$. $\mathcal{B}$, $\mathcal{E}^+$, $\mathcal{E}^-$ and $\mathcal{H}$ are all logic programs. The conditions for construction of $\mathcal{H}$ are:

Necessity: $\mathcal{B} \not\models \mathcal{E}^+$

Sufficiency: $\mathcal{B} \wedge \mathcal{H} \models \mathcal{E}^+$

Consistency: $\mathcal{B} \wedge \mathcal{H} \wedge \mathcal{E}^- \not\models \Box$

2.3.2 Terminology

This section introduces some logic programming terminology used throughout this document. More complete definitions can be found in the book *Foundations of Logic Programming* [Llo84].

A first-order alphabet consists of variables, predicate symbols and function symbols. 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 astring 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. $f(g(X),h)$ 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. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atomic formula. Both an atomic formula and its negation are literals. An atomic formula not negated is a positive literal and a negated atomic formula is 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 each X_j is a variable occuring in $L_1 \vee L_2 \vee ... \vee L_m$. In a clause there are no free variables: all variables are universally quantified. Since logical disjunction is commutative, we may represent a clause as a set of literals. For instance, the set $\{L_1, L_2, .., \neg L_i, \neg L_{i+1}, ...\}$ stands for the clause $(L_1 \vee L_2 \vee .. \neg L_i \vee \neg L_{i+1} \vee ...)$, which has an equivalent representation in $L_1 \vee L_2 \vee .. \vee \leftarrow L_i \wedge L_{i+1} \wedge ...$ [2] or, most commonly, $L_1, L_2, .. \leftarrow L_i, L_{i+1}, ...$, where commas on the left-hand side of $\leftarrow$ denote disjunctions, and commas on the right-and side denote conjunctions. A set of clauses is a clausal theory and represents the conjunction of its clauses. A well-formed formulae (*wff*) is a literal, a clause or a clausal theory. Whenever the set of variables in a wff E is empty, E is said to be ground. A Horn clause is a clause which contains at most one positive literal, and a definite program clause is a clause which contains exactly one positive literal. The positive literal is called the head of the clause and the conjuntion of negative literals is called the body of the clause. A Horn clause with no positive literals is a definite goal. A definite program clause with an empty body is a positive unit clause, or a fact, in Prolog [Bra00].

A definite program is a set of definite program clauses. Prolog allows literals of the form *notL*, where L is an atom, and *not* is interpreted under the *negation-as-failure* rule [Cla78]. Taking that into account, a program clause is a clause of the form:

$$T \leftarrow L_1, ..., L_m$$

where T is an atom, and each of L_i is of the form L or *notL*, where L is an atom. A normal program is, therefore, a set of program clauses. A predicate definition is a set of program clauses with the same predicate symbol in the head literal. The following two clauses define the predicate $daughter/2$:

$$\begin{array}{lcl} daughter(X,Y) & \leftarrow & female(X), mother(Y,X) \\ daughter(X,Y) & \leftarrow & notmale(X), father(Y,X) \end{array}$$

A constrained clause is a program clause in which all variables in the body also appear in the head. A non-recursive clause is a program clause in which the predicate symbol in the head does not appear in any of the literals in the body.

[2] $A \vee \neg B \equiv A \leftarrow B$

A predicate in a logic program is essentially the same as a relation in a database. The main difference between program clauses and database relations is in the use of types. A clause is typed if each variable appearing in the arguments of clause literals is constrained to a set of values. This association of predicates with database relations allows a formal representation of relations in Prolog, both extensional and intensional, as used in deductive databases [Llo84, Ull88].

2.3.3 Empirical ILP

The task of empirical ILP, which is concerned with learning a single predicate from a given set of examples, is formulated as follows [LD94]:

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 definion of predicate *p*,
- 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 $\mathcal{B}$.

p is the definition of the target relation. When learning from noisy examples, the completeness and consistency criteria need to be relaxed to avoid overfitting [LD92].

2.3.4 Interactive ILP

The problem of interactive ILP, as introduced by De Raedt [Rae92], is typically incremental and can be formalized as follows [LD94]:

Given:

- a set of traning 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 $\ominus$,
- 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 $\mathcal{B}$.

It is important to note that interactive ILP deals with the issue of representation change, i.e., shift of bias. Several interactive ILP systems can change the hypothesis language $\mathcal{L}$ during the learning process.

2.3.5 Structuring the hypothesis space

As previously noted, concept learning can be viewed as a search problem [Mit82]. Each state in he search space is a concept description 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.

In ILP the search space is the language of logic programs $\mathcal{L}$ consisting of program clauses of the form $T \leftarrow Q$, where T is an atom $p(X_1,...,X_n)$ and Q is a conjunction of literals $L_1,...,L_m$. The vocabulary of predicate symbols in the body of clauses is determined by the predicates from the background knowledge $\mathcal{B}$. Different ILP systems apply different language bias $\mathcal{L}$ as to syntactically restrict the form of clauses which can be formulated from a given vocabulary of predicates, function symbols and constants of the language. Such a restriction is relevant to reduce the search space to manageable forms.

A way to search the space of program clauses systematically can be achieved by introducing a partial ordering into a set of clauses based on the θ-subsumption. Clause c is said to $\theta - subsume$ clause c' if there exists a substitution θ, such that $c\theta \subseteq c'$ [Plo70]. Two clauses c and d are $\theta - subsumption\ equivalent$ if $c\ \theta - subsumes\ d$ and $d\ \theta - subsumes\ c$. A clause is reduced if it is not θ-subsumption equivalent to any proper subset of itself. A substitution $\theta = \{X_1/t_1,...,X_k/t_k\}$ 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 [Llo84].

With the use of θ-subsumption, the syntactic notion of generality is introduced in the search problem. Clause c is at least as general as clause c' ($c \preceq c'$) if $c\ \theta - subsumes\ c'$. Clause c is more general than c' ($c \prec c'$) if $c \preceq c'$ holds and $c' \preceq c$ does not. Since c

is a generalization of c', c' is a specialization (refinement) of c. θ-subsumption not only introduces the notion of generality, it also has two important properties. The first one is that if c θ-subsumes c' then c logically entails c' ($c \models c'$). The second one is that the relation $\preceq$ (at least as general as) introduces a lattice on the set of reduced clauses. Any two clauses have a least upper bound (*lub*) and a greatest lower bound (*glb*). Both the *lub* and *glb* are unique up to a renaming of variables.

The lattice on the set of reduced classes leads to the definition of 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 (*lub*) of c and c' in the θ-subsumption lattice. Since θ-subsumption and least general generalization do not take into account any background knowledge, their computation is simple and easy to implement in an ILP system. Semantic generality [Nib88, Bun88] (clause c is at least as general as clause c' with respect to background theory $\mathcal{B}$ if $\mathcal{B} \cup \{c\} \models c'$) is general undecidable and does not introduce a lattice on a set of clauses. Syntactic generality (using the θ-subsumption operation) is therefore more frequently used in ILP systems.

θ-subsumption not only provides a structure to hypothesis space, it can also be used to prune it. When generalizing c to c' ($c' \prec c$), all the examples covered by c will also be covered by c'. In case c is inconsistent, all its generalizations will also be inconsistent, thus not requiring any further consideration. When specializing c to c' ($c \prec c'$), an example not covered by c will not be covered by any of its specializations either. If c does not cover a positive example none of its specializations will, hence not requiring any further consideration. θ-subsumption therefore provides the basis for the important ILP techniques: bottom-up building of least general generalizations, and top-down searching of refinement graphs.

2.3.6 Generalization techniques

Generalization techniques search the hypothesis space in a bottom-up manner: they start from the training examples (most specific hypothesis) and search the hypothesis space by using generalization operators. Generalization techniques are best suited for interactive and incremental learning from few examples.

In bottom-up hypothesis generation, a generalization c' of clause c ($c' \prec c$) can be obtained by applying a θ-subsumption-based generalization operator. A generalization operator p maps a clause c to a set of clauses $p(c)$ which are generalizations of c. The syntactic operations this kind of operators perform on a clause are applying an inverse substitution to the clause and/or removing a literal from the body of the clause.

2.3.6.1 Relative least general generalization

The notion of least general generalization [Plo70] forms the basis of cautious generalization algorithms which perform bottom-up search of the θ-subsumption lattice. The word cautious here is based on the assumption that if c_1 and c_2 are true, then it is very likely that $lgg(c_1, c_2)$ will also be true. The lgg of terms, atoms and literals is defined both in Plotkin's paper *A Note on Inductive Generalization* [Plo70] and in the book *Inductive Logic Programming: Techniques and Applications* [LD94].

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 $\mathcal{B}$. For example, 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:

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

To illustrate the use of *rlgg*, we'll use the Example 2.2 and GOLEM's [MF90] constraints on introducing new variables into the body of it, as the resulting clause can be intractably large. The *rlgg* of the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ is computed as follows. 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 is:

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

$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$ produces the following clause:

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

Eliminating the irrelevant literals yields for the clause:

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

which stands for

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

2.3.6.2 Inverse resolution

The idea of inverse resolution [MB88], a generalization technique, is to invert the resolution rule of deductive inference [Rob65]. The basic resolution step in propositional logic derives the resolvent $p \vee r$ given the premises $p \vee \neg q$ and $q \vee r$. While resolution in propositional logic is quite straightforward (see Figure 2.1, the same in first-order logic is more complicated, involving substitutions.

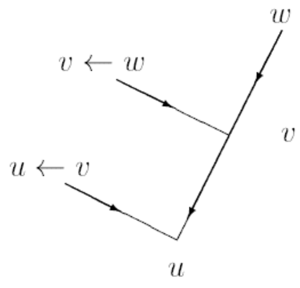


Figure 2.1: A simple propositional derivation tree

Inverse resolution inverts the resolution process using a generalization operator based on inverting substitution [Bun88]. 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 $W\theta\theta^{-1} = W$. Figures 2.2 and 2.3 illustrate this principle on the daughter example introduced in 2.2.

Example 2.3 (Inverse substitution)**.** Taking into account the Example introduced in 2.1, let $c = sibling(X,Y) \leftarrow parent(X,Z), parent(Y,Z)$ and the substitution $\theta = \{X/bob, Y/alice, Z/george\}$:

$$c' = c\theta = sibling(bob,alice) \leftarrow parent(bob,george), parent(alice,george)$$

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

$$c = c'\theta = sibling(X,Y) \leftarrow parent(X,Z), parent(Y,Z)$$

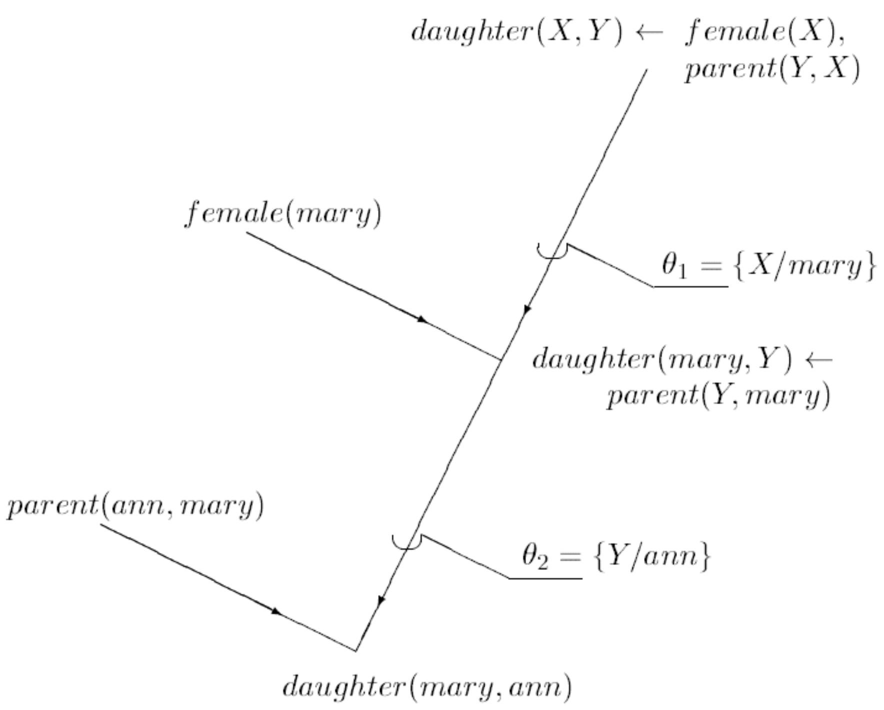


Figure 2.2: A first order linear derivation tree

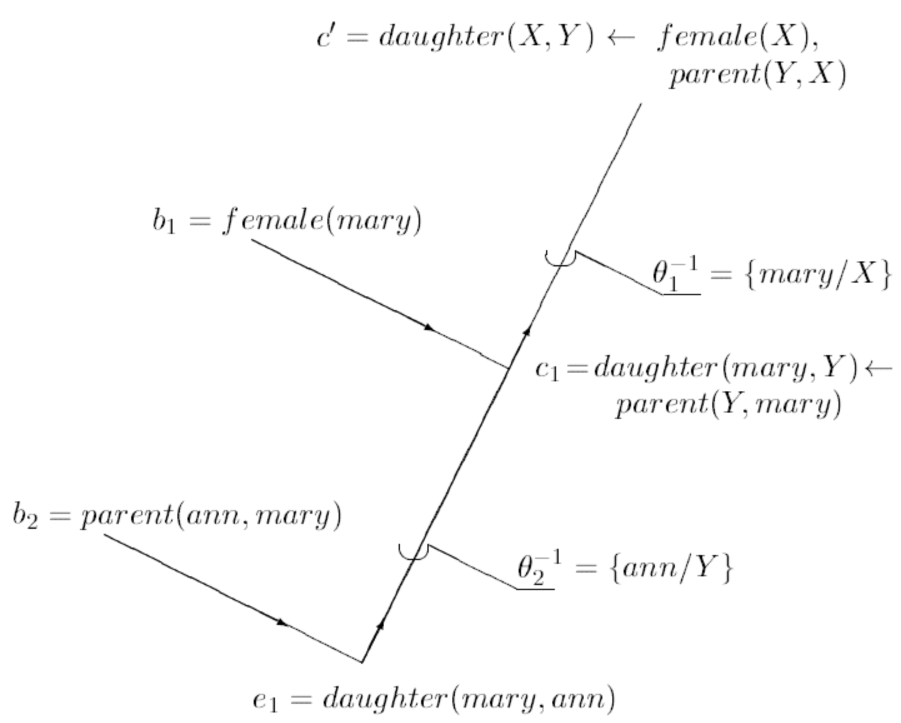
The general case is, however, substantially more complex, because it involves the places of terms in order to ensure that the variables in the initial wff W are appropriately restored in $W\theta\theta^{-1}$. A simple example to show the need of keeping record of places is available in wff $W = loves(X, daughter(Y))$ applied to the substitution $\theta = \{X/ann, Y/ann\}$. The inverse substitution needs to known the original variable places, as being simply defined as $\theta^{-1} = \{ann/X, ann/Y\}$ wouldn't suffice.

The steps in inverse resolution attempt to find clauses that, together with a clause from the background knowledge will entail the current positive example and therefore be added to the current hypothesis $\mathcal{H}$ instead of it. Each step has an inverse substitution associated with it.

The problem with inverse resolution is that it is non-deterministic, because at each inverse resolution step, various generalizations of a clause can be performed, depending on the clause chosen from the background knowledge and the employed inverse substitution.

2.3.7 Specialization techniques

Specialization techniques search the hypothesis space in a top-down manner, from general to specific hypothesis, using a θ-subsumption-based specialization operator. Such an operator is usually called a refinement operator, and maps a clause c to a set of clauses



$c' = daughter(X,Y) \leftarrow female(X), parent(Y,X)$
$b_1 = female(mary)$
$\theta_1^{-1} = \{mary/X\}$
$c_1 = daughter(mary,Y) \leftarrow parent(Y,mary)$
$b_2 = parent(ann,mary)$
$\theta_2^{-1} = \{ann/Y\}$
$e_1 = daughter(mary,ann)$

Figure 2.3: An inverse linear derivation tree

$p(c)$ which are specializations (refinements) of c. A refinement operator typically computes the set of most general specializations of a clause under θ-subsumption. The two basic syntactic operations on a clause are applying a substitution on a clause and adding a literal to the body o the clause.

Top-down learners start from the most general clauses and repeatedly specialize them until they no longer cover negative examples. During the search, it is ensured that at least one positive example is covered.

2.3.7.1 Top-down search of refinement graphs

The hypothesis space of a program clauses, restricted by language bias $\mathcal{L}$ and with respect to background knowledge $\mathcal{B}$ is a lattice structured by the θ-subsumption generality ordering. A refinement graph can be defined in this lattice, therefore enabling the search to be directed from general to specific hypothesis. A refinement graph is a directed, acyclic graph, where nodes represent program clauses and arcs corespond to the basic refinement operators.

The MIS algorithm [Sha05] defines an interactive algorithm for learning hypothesis, in the language of definite clauses and with respect to background knowledge predicates $\mathcal{B}$. The training examples are incrementally read. For each example read, a loop that checks for the completeness and consistency of the current hypothesis is entered. If the current hypothesis covers a negative example, then the incorrect clauses are deleted from $\mathcal{H}$. If, on the other hand, there is a positive example not covered by the current hypothesis, then the refinement graph is searched to develop a clause which covers it. The previous two steps are made until $\mathcal{H}$ is both complete and consistent.

Example 2.4 (Searching the refinement graph)**.** To illustrate the above process, the example introduced in 2.1 shall be used for the building of a refinement graph. For simplicity, the language $\mathcal{L}$ is restricted to non-recursive definite clauses and the refinement operator only uses a specialization operation: adding a literal to the body of a clause. The top-level node in the refinement graph is formally the most general clause $false$. However, in practice, the search starts with the most general definition of the predicate *sibling*:

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

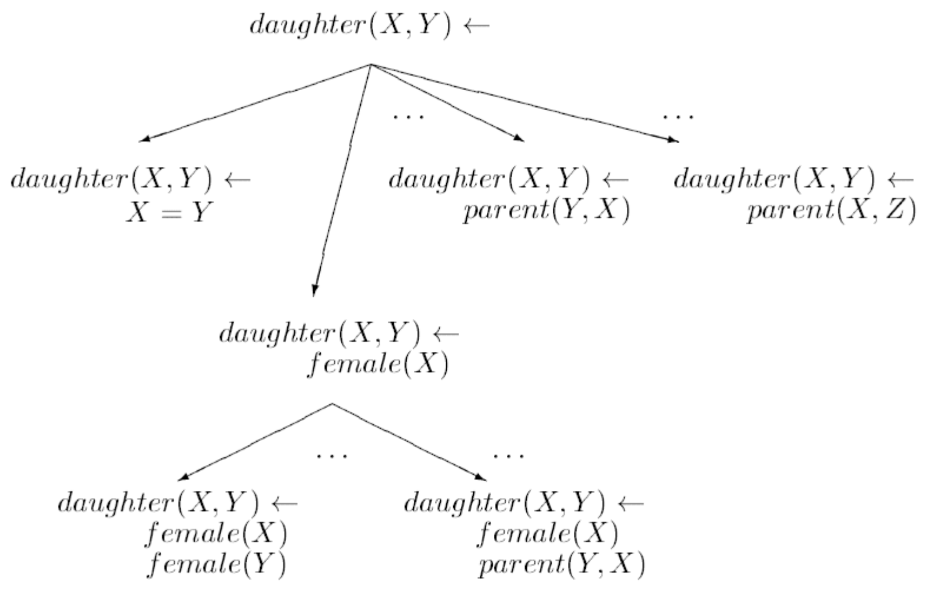
Here, an empty body is written instead of the body $true$. The learner is first given the positive example $e_1 = sibling(bob, alice)$. Since c covers e_1, the current hypothesis is initialized to $\mathcal{H} = \{c\}$. The second example $e_2 = sibling(alice, bob)$ is also positive and covered by $\mathcal{H}$, hence not resulting in a specialization of the hypohsis. The third example $e_3 = sibling(harriet, george)$ is, however, negative. Singe c covers this example, it is deleted from $\mathcal{H}$ and the process of generating a new clause that covers the first positive examples is initialized. The learner therefore generates the set of refinements (minimal specializations) of clause c, by applying the refinement operator p. The set of new clauses take the form:

$$p(c) = \{sibling(X,Y) \leftarrow L\}$$

where L is one of the following (note that the language is restricted to definite and non-recursive clauses):

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

The refinements obtained are then considered one by one, being retracted whenever they don't cover the positive example e_1. The refinements which descriminate between positive and negative examples are retained in the current hypothesis. Such refinements of c are $sibling(X,Y) \leftarrow parent(X,Z)$ and $sibling(X,Y) \leftarrow parent(Y,Z)$. Since they are not



daughter(X, Y) ←
daughter(X, Y) ← X = Y
daughter(X, Y) ← parent(Y, X)
daughter(X, Y) ← parent(X, Z)
daughter(X, Y) ← female(X)
daughter(X, Y) ← female(X) female(Y)
daughter(X, Y) ← female(X) parent(Y, X)

Figure 2.4: Part of the refinement graph for the daughter relations problem

consistent with respect to all the negative examples, they are further refined, eventually leading to clause $sibling(X,Y) \leftarrow parent(X,Z), parent(Y,Z)$ which is both complete and consistent.

Figure 2.4 shows part of the refinement graph for the daughter problem introduced in example 2.2.

2.4 Advanced topics on ILP

2.4.1 ILP as search of refinement graphs

As previously stated, the learning task can be seen as a search task in the space of hypothesis [Mit82]. The learning techniques introduced in section 2.3 define a structure on the search space via the θ-subsumption operator, using either generalization or specialization operators. The use of refinement operators define a refinement graph. Search can be directed from general to specific (in a *top-down* manner) or from specific to general (in a *bottom-up* manner). This kind of structure enables the use of different search strategies. While *best-first search* [Pea84] is usually the desired search strategy, time complexity (since the search space can grow very fast) usually requires the choice for a simpler strategy, such as *beam search* [Low76] or *hill-climbing* [RN02].

There are two types of search heuristics: heuristics which direct the search and heuristics which decide when to stop the search, called *stopping criteria*. Two types of stopping criteria can be distinguished. The first (the *necessity stopping criterion*) determines when to stop building a clause. In exact domains, this usually requires consistency with respect to the set of training examples and background knowledge. The second one (the *sufficiency stopping criteria*) determines when to stop building new clauses for the target predicate definition. Usually the search is terminated when all positive examples have been covered. However, to achieve noise tolerance, avoid overfitting and obtain more compact concept descriptions, an alternative stopping criteria may be applied, usually of statistical nature.

Refinement graphs, as a result of the application of refinement operator p, induce a generality ordering on $\mathcal{L}$. That is, they itroduce the relation 'is more general' with respect to a given refinement operator p, denoted by $\prec_p$. c is more general than c' ($c \prec_p c'$) if there exists a directed path from c to c' in the refinement graph induced by p.

The cost of searching a refinement graph depends on the node evaluation cost (*NEC*), which may not be constant, the branching factor ($BF(c)$) of nodes in the graph and on the length of the clause generated (*LOC*). Different search methods may contribute with more factors, such as the beam width [RN02] when using beam search.

2.4.1.1 Mode-Directed Inverse Entailment

Since its appearance as a research topic, ILP research has spawned various theoretical topics, namely the problem of inverting resolution [MB88, Wir89, Rou92], inversion of clausal implication [LM92, IA93, MP94], predicate invention [Mug94], closed world specialisation [BM91] and U-learnability [MP98]. Stephen Muggleton took that into account and proposed a generalisation and enhancement of previous approaches in Mode Directed Inverse Entailment (*MDIE*) [Mug95]. Muggleton demonstrated that the problem of inverse resolution could be made simpler with an approach from the direction of model theory rather than resolution proof theory.

MDIE builds on the notion of a bottom-clause. To explain what a bottom-clause is, it is relevant to consider the general problem specification of ILP. That is, given background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ find the simples consistent hypothesis $\mathcal{H}$ such that:

$$\mathcal{B} \wedge \mathcal{H} \models \mathcal{E}$$

In case $\mathcal{H}$ and $\mathcal{E}$ are single Horn clauses, the previous equation can be rearranged (through the notion of absortion [MB88]) to give:

$$\mathcal{B} \wedge \neg\mathcal{E} \models \neg\mathcal{H}$$

$\mathcal{B}$	$\mathcal{E}$	$\mathcal{K}$
$anim(X) \leftarrow pet(X).\ pet(X) \leftarrow dog(X).$	$nice(X) \leftarrow dog(X).$	$nice(X) \leftarrow dog(X), pet(X), anim(X).$
$hasbeak(X) \leftarrow bird(X).\ bird(X) \leftarrow vulture(X).$	$hasbeak(tweety)$	$hasbeak(tweety); bird(tweety); vulture(tweety).$
$white(swan1).$	$\leftarrow black(swan1)$	$\leftarrow black(swan1), white(swan1)$
$sentence([],[]).$	$sentence([a,a,a],[]).$	$sentence([a,a,a],[]) \leftarrow sentence([],[]).$

Table 2.3: The most specific clause for various versions of background knowledge and example.

Let $\neg\mathcal{K}$ be the (potentially infinite) conjunction of ground literals which are true in all models of $\mathcal{B} \wedge \neg\mathcal{E}$. Since $\neg\mathcal{H}$ must be true in every model of $\mathcal{B} \wedge \neg\mathcal{E}$ it must contain a subset of ground literals in $\neg\mathcal{K}$. Therefore:

$$\mathcal{B} \wedge \mathcal{E} \models \neg\mathcal{K} \models \neg\mathcal{H}$$

and so, for all H:

$$\mathcal{H} \models \mathcal{K}$$

This leads to the observation that a subset of the solutions for $\mathcal{H}$ can be found by considering the clauses which θ-subsume $\mathcal{K}$. The complete set of candidates for $\mathcal{H}$ can be found by considering all clauses which θ-subsume sub-saturants of $\mathcal{K}$, thus significantly reducing the search space.

Table 2.3, taken from [Mug95], show most specific clauses for different versions of background knowledge and example.

As $\mathcal{K}$ can have infinite cardinality, mode declaration [Mug95] is used to constrain the search of clauses which θ-subsume $\mathcal{K}$. A mode declaration has either the form $modeh(n,atom)$ or $modeb(n,atom)$ where n, the recall, is either an integer, $n > 1$, or '*' and *atom* is a ground atom. Terms in the *atom* are either normal or placemarker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A placemarker is either $+type$, $-type$ or #*type*, where type is a constant. If m is a mode declaration, then $a(m)$ denotes the atom of m with placemarkers replaced by distinct variables. The sign of m is positive if m is a *modeh* and negative if m is a *modeb*. The recall is used to bound the number of alternative solutions for instantiating the atom. The following are mode declarations:

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

MDIE-based ILP systems usually introduce a bias on the depth of variables in the definite mode language. MDIE has become widely used in ILP systems, being a significant enhancement to previous top-down learning methods. For that matter, the general induction procedure is presented. The *induce*() procedure starts from a set of examples E, a background knowledge B, and some constraints C, following a greedy cover set approach to induce a theory H. The *saturation* step builds the most specific clause, a definite clause that is used to constrain the search space. The *search*() procedure works top-down, by adding literals from the botom-clause to an existing clause. Most of the work is done in this procedure, which is constrained to a given depth in C in order to ensure its termination. If no hypothesis is found, then e_i is returned. The loop continues by removing the examples covered by the new rule and searching for a clause that covers the remaining examples.

Algorithm 1 A general induction procedure based on MDIE

```
$i = 0$
$H_i = \emptyset$
**loop**
  **if** $E^+ = \emptyset$ **then**
    return $H_i$
  **end if**
  $i = i + 1$
  $Train = E^+ \cup E^-$
  $e_i^+ = select\ an\ example\ from\ E^+$
  $\perp_i = saturate(B, C, H_{i-1}, e_i^+)$
  $h_i = search(B, C, Train, H_{i-1}, e_i^+, \perp_i)$
  $H_i = H_{i-1} \cup h_i$
  $E_{covered} = \{e | e \in E^+ \wedge B \cup H_i \models e\}$
  $E^+ = E^+ E_{covered}$
**end loop**
```

2.4.2 Handling imperfect data in ILP

Real-world data is almost never perfect. For that matter, it is important to identify diferent kinds of imperfect data and be able to apply some efficient noise-handling mechanisms. As previously stated, imperfections in data include noise, too sparse training examples, inexacteness of the description language and missing values. Learning systems usually have a single mechanism for dealing with all but the missing values problem, often called *noise-handling mechanisms*. Such mechanisms prevent the induced hypothesis from over-fitting the data set. Dealing with missing values usually requires a different approach.

2.4.2.1 Handling missing values

The problem of handling missing values from training examples is more relevant in an attribute-value framework, as in ILP some characteristics can easily be omitted. However, some ILP systems are based on attribute-value learners, so introducing this problem is somewhat relevant for the purpose of this document.

Usually, learning systems are not able to handle missing data on their known. The problem has to be solved by the knowledge engineer before running the learning system. This happens because there is no silver-bullet in handling missing values [HKP06, WF05]. The most frequent way to solve the problem is by replacing a missing value by the majority value of the attribute within the class. Alternative ways include considering the unknown value as a legal value or using a classifier on the rest of the attributes to determine the most likely value. A more sophisticated approach used in decision-tree learning systems like ASSISTANT [CKB87] or rule induction learning systems like CN2 [CN89, CB91] is to replace the missing value with several examples, one for each of the possible attribute values, weighted by the conditional probabilities of them.

The problem of handling missing values in ILP is usually more relevant when there is a lack of examples. LINUS [DL91, LD94] provides four options of negative example generation, based on ideas from QuMAS [Moz87]:

- Negative facts can be given explicitly.
- When generating negative facts under the *closed-world assumption*, all possible combinations of values of the n arguments of the target predicate are generated.
- Under the *partial closed-world assumption*, 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 $n = n_{Dep} + n_{Ind}$.

2.4.2.2 Handling noise

Most modern attribute-value learning programs contain mechanisms to deal with noisy data. Those belonging to the Top Down Induction of Decision Trees [Qui86] family of programs handle noise with a *tree pruning* mechanism. Induction programs such as

AQ15 [MMHL86] and CN2 [CN89, CB91] use mechanisms such as *rule truncation* or *significance tests*, respectively.

The mechanisms for dealing with noise in ILP can be based in appropriate search heuristics and stopping criteria used in hypothesis construction. An alternative approach is by post-processing the hypothesis found by considering data completely correct.

One possible stopping criteria is based on the *encoding length restriction* [Qui90], which restricts the total length of an induced clause to the number of bits needed to explicitly enumerate the positive examples it covers. The number of bits needed to explicitly indicate $n^{\oplus}(c)$ positive examples covered by clause c out of the n_{cur} examples in the current training set is:

$$ExplicitBits(c, \mathcal{E}_{cur}) = log_2(n_{cur}) + log_2(\frac{n_{cur}}{n^{\oplus}(c)})$$

The number of bits needed to encode a clause with m literals in the body is calculated as:

$$ClauseBits(c) = \sum_{i=1}^{m}(1 + log_2(l) + log_2(V_{q_i})) - log_2(m!)$$

where l is the number of different predicates in the background knowledge and V_{q_i} is the number of possible choices of variables of the predicate used in literal L_i [Qui90].

The construction of a clause is stopped (the necessity stopping criteria is satisfied) when no negative examples are covered by the clause or when adding any literal with positive gain would cause $ClauseBits(c)$ to exceed $ExplicitBits(c, \mathcal{E}_{cur})$.

The construction of a hypothesis is terminated (the sufficiency stopping criteria is satisfied) when all the positive examples are covered or when all the literals with positive gain require more than $ExplicitBits(c, \mathcal{E}_{cur})$ to encode.

The encoding length restriction has some deficiencies, as it may prevent the building of complete definitions in non-noisy domains and allows very specific clauses covering a small number of examples. A more sophisticated encoding scheme, which takes into account the information necessary to encode the background knowledge as well as the proofs which would generate the training examples covered is introduced in [MSB92].

There have been attempts to introduce pruning in ILP, splitting the training set into a set for learning and a set for pruning. Operators that delete the last literal and drop a clause are independently applied to each clause of the induced hypothesis, being retained the modification that yields the greatest accuracy. The procedure is repeated until a decrease in accuracy is observed.

Some heuristics have been proposed to guide the search of literals [LCD92a, LCD92b]. This same heuristics can be used as stopping criteria, deciding whether to stop adding literals to the clause and whether to stop adding clauses to hypothesis. In addition to the entropy/information-based search heuristics used in the construction of decision trees, a variety of other heuristics have ben proposed by other authors [BFSO84, Min89]. Heuristics use different probability estimates. It was shown that the method of estimating probabilities used in the heuristics has a greater impact on the accuray of the induced hypothesis than the actual form of the heuristics [Ces91].

The following measures assume that n is the number of examples in the initial training set, $n^{\oplus}$ of which are positive and $n^{\ominus}$ negative. n_{cur} denotes the number of examples in the current training set $\mathcal{E}_{cur}$, and $n(c)$ the number of examples in the local training set $\mathcal{E}_c$ (that is, the set of examples from the current training set $\mathcal{E}_{cur}$ which are covered by clause c). The *quality* of clause c is computed by estimating the *goodness of split*, i.e., how good is the distribution $n^{\oplus}(c) - n^{\ominus}(c)$ of positive and negative examples covered by the clause.

The *expected classification accuracy* is defined as the probability that an example covered by clause c is positive:

$$A(c) = p(\oplus|c)$$

The *informativity* is the amount of information necessary t specify that an example covered by the clause is positive:

$$I(c) = -log_2 p(\oplus|c)$$

Given a new clause c', the *accuracy gain AG* and *information gain IG* are defined as follows:

$$AG(c',c) = A(c') - A(c) = p(\oplus|c') - p(\oplus|c)$$

$$IG(c',c) = I(c) - I(c') = log_2 p(\oplus|c') - log_2 p(\oplus|c)$$

Accuracy gain and information gain can be weighted by the reltive frequency of positive examples covered at an individual specialization step, leading to the measures of *weighted accuracy gain WAG* and *weighted information gain WIG*:

$$WAG(c',c) = \frac{n^{\oplus}(c')}{n^{\oplus}(c)} \times (p(\oplus|c') - p(\oplus|c))$$

$$WIG(c',c) = \frac{n^{\oplus}(c')}{n^{\oplus}(c)} \times (log_2 p(\oplus|c') - log_2 p(\oplus|c))$$

The previous heuristics for evaluating the quality of a clause use probabilites that need to be estimated from the current training set $\mathcal{E}_{cur}$. Relative frequency is often used as an estimation:

$$p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)}$$

The laplace estimate is more reliable than relative frequency when dealing with a small number of examples, but relies on the assumption of a uniform prior probability distribution of the classes [NB86]. The laplace estimate is defined as follows:

$$p(\oplus|c) = \frac{n^{\oplus}(c)+1}{n(c)+2}$$

The *m*-estimate [Ces90] avoids the problems of relative frequency an the Laplace estimate, by taking into account the prior probabilities of the classes. The prior probability $p_a(\oplus)$ can be estimated by the relative frequency. The parameter *m* reflects the confidence in the experimental evidence, hence can be set subjectively. The *m*-estimate is defined as follows:

$$p(\oplus|c) = \frac{n^{\oplus}(c) + m \times p_a(\oplus)}{n(c)+m}$$

2.4.3 Efficiency in ILP systems

ILP systems often require long running times and large amounts of memory to produce valuable models. Improving efficiency of ILP systems has thus been recognized as one of the main issues to be addressed by the ILP community [PSCF03, FCR+09]. ILP algorithms are sufficiently complex to offer a large scope for research on improvements. Initial research on improving the efficiency of ILP systems focused on reducing the search space [Cam02, NRB+96] and efficiently testing candidate hypothesis [BDD+02, CSC+02].

Some notions of declarative bias for reducing the search space have already been previously introduced, so the focus of this section is on methods to efficiently test candidate hypothesis (namely on the coverage mechanism), some specialized storage mechanisms and possible improvements on the inference engine. The choice for this particular methods is related to their orthogonality, since they can easily be integrated in an existing ILP system.

2.4.3.1 Evaluation of hypothesis coverage

In ILP systems, hypothesis are generated usually be extending a previous clause, and then evaluated by computing how many examples they cover. Generating a new clause is often straightforward. The evaluation of individual hypothesis is the hard part [BGSS00], specially if the number of examples is large. Research on improving the evaluation of hypothesis has been related to query transformations [CSC00, CSC$^+$02], query-packs [BDD$^+$02] and lazy evaluation of examples [Cam03, FCR$^+$09].

Query transformations apply some systematic techniques to, while retaining the set of examples derived from the hypothesis, reduce the number of computations necessary to determine coverage of examples. This kind of transformations can take into account redundancy in predicates on the body of the hypothesis, apply intelligent cuts (!) to goals whose solutions for previous predicates in the body of the clause do not alter their computation, and use previous exploration of “parent” clauses (those that originate the current clause, by specification), simplifying derivability tests.

Query-packs group similar queries in an unique set, in a tree-like structure. However, query-packs, to be efficiently executed, require changes at the level of the Prolog engine itself, introducing appropriate primitives for a more efficient procedural semantics of the *or* operator.

Lazy evaluation of examples is a technique that tries to avoid unneessary use of examples in the coverage computations. Lazy evaluation does not affect the completeness of the hypothesis search. Lazy evaluation of hypothesis is usually distinguished between lazy evaluation of negative examples, lazy evaluation of positive examples and total laziness. Lazy evaluation of negative examples builds on the observation that we are only interested in knowing if the hypothesis covers more than the llowed number of negative examples or not. One is not really interested to know how much more negative examples a hypothesis covers than allowed. When using lazy evaluation of positives, we start by determining if an hypothesis covers more positives than the current best consistent hypothesis. If it does, the positive examples are evaluated just until the best cover so far is exceeded. If the best cover is exceeed, the hypothesis is retained, otherwise it can be discarded. Total laziness goes a bit further and simply does not evaluate the positive cover at all for partial clauses (clauses that are inconsistent with the negative examples).

2.4.3.2 Storage mechanisms

ILP systems search large spaces of hypothesis. In larger applications this space can quickly grow to thousands and even millions of different clauses. It is observable that the ILP search space is highly redundant [NdW97], as the same clause can be proposed over and over again and quite often the clauses are very similar, or share a common prefix. Tries were originially invented by Fredkin [Fre60] to index dictionaries and have since been generalized to index recursive data structures such as terms. Tries have been extensively used in automated theorem proving, term rewriting and tabled logic programs [BCR93, Gra96, McC92, Ohl90, RRS$^+$99]. The basic idea behind the trie data structure is to partition a set of terms based upon their structure so that looking up and inserting can be performed in a single pass through a term.

ILP systems often keep an *open list* of clauses that deserve to be refined, or further explored. For each such clause it is quite convenient to keep a list of examples covered, the *coverage list*. This helps when computing coverage for the new clauses. Coverage lists represent sets of examples. However, they do not have the nice incremental property of clauses. The ability to represent sparse spaces addressed by the generic data structures called quadtrees [Sam84] can be extended to the coverage lists. Quadtrees are hierarchical data structures based on recursive decomposition of space. The RL-tree (RangeList-Tree) [FCR$^+$09] data structure is a quadtree based data structure suitable for a single dimension space. It allows the storage of lists of intervals representing integer number. For example, the list $[1,2,5,6,7,8,9,10]$ is represented as the list of intervals $[1-2,5-10]$. Since the efficiency of basic operations on interval lists is linear on the number of nodes, RL-trees try to achieve efficient storage and manipulation of a coverage list by a recursive decomposition of intervals.

2.4.3.3 Performance of the inference engine

Ideally, ILP systems should spend most of their running time performing inference. Most often, they use the inference mechanism of a Prolog engine to do theorem proving. As a result, it becomes easier to implement the whole ILP system in Prolog. However, as ILP systems address larger applications, they challenge the traditional Prolog engines. Therefore, *indexing* becomes critical for good ILP performance [CSL07]. Indexing can be implemented both for static data structures, representing the databases, and dynamic data structures representing the search space. The following indexing strategies follow the ones implemented in the Yap Prolog system [CDRA, CSL07].

Many modern Prolog systems follow D. Warren's work on the Warren Abstract Machine (WAM) [War83] and implement a simple form of static indexing which relies on the main functor of the first argument to select a subset of all clauses, which are then tried one by one. This traditional WAM indexing shows several limitations for ILP systems. First, ILP systems do not always have the first argument bound. On the other hand, generating indexing code for every argument would be extremely space-extensive [PN91, CDWY96]. *Just-in-time indexing* applies the Just-In-Time compilation strategy [Ayc03] so that all the required indices are generated, and only the ones required. Indexes are generated based on the instantiation of the current goal and expanded to give different instantiations for the same goal [FCR$^+$09].

Dynamic indexing comes to consideration when dealing with update operations such as *assert* or *retract*. One alternative would be to destroy he indexing tree every time the database is changed. This, however, may lead to a generation of the entire tree in every query. A less expensive solution is to manipulate the indexing tree in a way that it is always consistent with the database. This approach walks the indexing tree trying to find the minimal set of expansions to a tree, inserting clauses in unbound chains in case of asserts and reverting insert operations in case of retracts.

2.4.4 Parallel and distributed ILP

Parallel implementations of ILP systems aim to obtain an algorithm with a speedup proportional to the number of processors over the best available serial algorithm. The central issue in designing a system to support parallellism is how to break up a given task into subtasks, each of which will be executing in parallel with the other [Hwa89, Sil09]. The problem posed is then to partition the ILP task into subtasks. The following definitions can be derived from the ILP problem general specification [DR95]:

Definition 1. *A partition $\{T_1, ..., T_n\}$ of an ILP-task $T = (\mathcal{B}, \mathcal{E}, \mathcal{L})$ is a set of ILP-tasks $T_i = (\mathcal{B}, \mathcal{E}_i, \mathcal{L}_i)$ such that $\mathcal{E}_i \subset \mathcal{E}$, and $\mathcal{L}_i \subset \mathcal{L}$ for all i, and that $\cup_{i=1}^{n} \mathcal{E}_i = \mathcal{E}$ and $\cup_{i=1}^{n} \mathcal{L}_i = \mathcal{L}$.*

Definition 2. *A partition $\{T_1, ..., T_n\}$ of an ILP-task T is valid if and only if the union $\cup_{i=1}^{n} \mathcal{H}_i$ of partial hypothesis $\mathcal{H}_i$ obtained by applying a common sequential ILP algorithm A to task T_i is equivalent to the solution hypothesis $\mathcal{H}$ obtained by applying algorithm A to task T.*

Definition 2 can be applied to two different notions of explanation and semantics currently distinguished in ILP: the *normal ILP setting* and the *nonmonotonic ILP setting* [RL93, MdR94]. This difference has lead to some constraints on parallel ILP [DR95], namely that the partitioning is only permitted, in a normal setting, if one is learning a single predicate without recursion. For the nonmonotonic setting, valid partitions of the ILP task can be produced by splitting the language bias $\mathcal{L}$. Experiments on the knowledge discovery system *Claudien* [RB93, DR95] showed that linear speedup is possible, at least for low degrees of concurrency, in a nonmonotonic setting.

2.5 Applications of ILP

ILP systems have been successfully used in both industrially and scientifically relevant problems. An important application of ILP is knowledge acquisition in second generation expert systems, which use a first-order represenation a deep model of the domain. ILP is also one of the foundations of relational data mining [DL01], being actively used in knowledge discovery in databases. Finally, ILP can be used as a tool in various steps of sicentific discovery.

Some of the most relevant applications of ILP in real-world data include learning qualitative models from sample behaviors [BMV91, DB92], inducing temporal rules for satellite fault diagnosis [Fen91], predicting secondary structure of proteins [MKS92] and finite element mesh design [DM92b, Dol91, DD91].

Chapter 3

ILP Systems

This chapter introduces some of the most relevant ILP systems up to date. Some of the descriptions here were are taken from [LWZ$^+$96, Dze].

3.1 An overview of FOIL

FOIL is a system for learning intensional concept definitions from relational tuples. It is one of the best-known and successful empirical ILP systems and has inspired a lot of further research. The following description was taken from [LWZ$^+$96].

FOIL [Qui90, QC93] induces concept definitions represented as function-free Horn clauses, optionally containing negated body literals. The background knowledge predicates are represented extensionally as sets of ground tuples. FOIL employs a heuristic search strategy which prunes vast parts of the hypothesis space. As its general search strategy, FOIL adopts a covering approach. Induction of a single clause starts with a clause with an empty body which is specialised by repeatedly adding a body literal to the clause built so far. As candidate body literals, FOIL considers the literals which are constructed by variabilising the predicates (including the target predicate), that is, by distributing variables to their argument places. Additionally, FOIL takes into account literals stating (in)equality of variables in the head or body literals. Furthermore, literals may contain constants which the user has declared as theory (i.e. relevant) constants. All literals have to conform to the type restrictions of the predicates. For further control of the language bias, FOIL provides parameters limiting the total number and maximum depth of variables in a single clause. In addition, FOIL incorporates mechanisms for excluding literals which might lead to endless loops in recursive hypothesis clauses. FOIL offers limited number handling capabilities by generating literals which compare numeric variables to each other or to thresholds. Among the candidate literals, FOIL selects one literal to be added to the body of the hypothesis clause according the information gain heuristic, an information-based measure estimating the utility of a literal in dividing positive from negative examples. FOIL stops adding literals to the hypothesis clause if the clause reaches a predefined minimum accuracy or if the encoding length of the clause exceeds the number of bits needed for explicitly encoding the positive examples it covers. This second stopping criterion prevents the induction of overly long and specific clauses in noisy domains. Induction of further hypothesis clauses stops if all positive examples are covered or if the set of induced hypothesis clauses violates the encoding length restriction. In a postprocessing stage, FOIL removes unneccessary literals from induced clauses as well as redundant clauses from the concept definition. FOIL's greedy search strategy makes it very efficient, but also prone to exclude the intended concept definitions from the search space. Some refinements of the hill-climbing search alleviate its short-sightedness, such as including a certain class of literals with zero information gain into the hypothesis clause and a simple backtracking mechanism.

FOIL is a batch learning system which reads in all learning input from a single input file. Positive as well as negative examples are required for learning. A user may provide negative examples explicitly or, alternatively, instruct FOIL to generate negative examples automatically according to the Closed World Assumption (CWA). In the latter case, the set of positive examples must be complete up to a certain example complexity. For predicates with high arity, the CWA may generate a huge number of negative examples. FOIL offers a command line option allowing the user to specify the percentage of randomly-selected negative examples to be used for induction.

Examples and background knowledge for FOIL have to be formatted as tuples, that is, each ground instance of a predicate is represented as a sequence of argument values. For each predicate, the user provides a header defining its name and argument types. Optionally, the user may indicate the input/output mode of the predicates, thus further limiting the number of candidate body literals. For convenient testing of the induced hypothesis, the user may provide test cases (i.e. classified examples) for the target predicates together with the learning input. FOIL then checks the hypothesis on these cases and reports the results.

3.2 An overview of GOLEM

The following description was taken from [LWZ^{+}96].

As FOIL, GOLEM [MF90] is a "classic" among empirical ILP systems. It has been applied successfully on real-world problems such as drug design [KMLS92], protein structure prediction [MKS92] and finite element mesh design [DM92a]. GOLEM copes efficiently with large datasets. It achieves this efficiency because it avoids searching a large hypothesis space for consistent hypotheses as, for instance, FOIL, but rather constructs a unique clause covering a set of positive examples relative to the available background knowledge. The principle is based on the relative least general generalisations (rlggs) introduced by Plotkin [Plo71, PMM71]. GOLEM embeds the construction of rlggs in a covering approach. For the induction of a single clause, it randomly selects several pairs of positive examples and computes their rlggs. Among these rlggs, GOLEM chooses the one which covers the largest number of positive examples and is consistent with the negative examples. This clause is then further generalised. GOLEM randomly selects a set of positive examples and constructs the rlggs of each of these examples and the clause obtained in the first construction step. Again, the rlgg with the greatest coverage is selected and generalised by the same process. The generalisation process is repeated until the coverage of the best clause stops increasing. GOLEM conducts a postprocessing step, which reduces induced clauses by removing irrelevant literals.

In the general case, the rlgg may contain infinitely many literals. Therefore, GOLEM imposes some restrictions on the background knowledge and hypothesis language which ensure that the length of rlggs grows at worst polynomially with the number of positive examples. The background knowledge of GOLEM is required to consist of ground facts. For the hypothesis language, the determinacy restriction applies, that is, for given values of the head variables of a clause, the values of the arguments of the body literals are determined uniquely. The complexity of GOLEM's hypothesis language is further controlled by two parameters, i and j , which limit the number and depth of body variables in a hypothesis clause.

GOLEM learns Horn clauses with functors. It may be run as a batch learner or in interactive mode where the induction can be controlled manually. GOLEM is able to learn from positive examples only. Negative examples are used for clause reduction in the postprocessing step, as well as input/output mode declarations for the predicates the user may optionally supply. For dealing with noisy data, GOLEM provides a system parameter enabling the user to define a maximum number of negative examples a hypothesis clause is allowed to cover.

3.3 An overview of MOBAL

MOBAL [MWKE93] is a system for developing operational models of application domains in a first order logic representation. The following description was taken from [Dze].

MOBAL integrates a manual knowledge acquisition and inspection environment, an inference engine, machine learning methods for automated knowledge acquisition, and a knowledge revision tool. By using MOBAL's knowledge acquisition environment, you can incrementally develop a model of your domain in terms of logical facts and rules. You can inspect the knowledge you have entered in text or graphics windows, augment the knowledge, or change it at any time. The built-in inference engine can immediately execute the rules you have entered to show you the consequences of your inputs, or answer queries about the current knowledge. MOBAL also builds a dynamic sort taxonomy from your inputs. If you wish, you can use several machine learning methods to automatically discover additional rules based on the facts that you have entered, or to form new concepts. If there are contradictions in the knowledge base due to incorrect rules or facts, there is a knowledge revision tool to help you locate the problem and fix it.

3.4 An overview of Progol

The following description was taken from [Dze].

PROGOL [Mug95] employs a covering approach like, e.g., FOIL. That is, it selects an example to be generalised and finds a consistent clause covering the example. All clauses made redundant by the found clause including all examples covered by the clause are removed from the theory. The example selection and generalisation cycle is repeated until all examples are covered. When constructing hypothesis clauses consistent with the examples, PROGOL conducts a general-to-specific search in the theta-subsumption lattice of a single clause hypothesis. In contrast to other general-to-specific searching systems, PROGOL computes the most specific clause covering the seed example and belonging to the hypothesis language. This most specific clause bounds the theta-subsumption lattice from below. On top, the lattice is bounded by the empty clause. The search strategy is an A^*-like algorithm guided by an approximate compression measure. Each invocation of the search returns a clause which is guaranteed to maximally compress the data, however, the set of all found hypotheses is not necessarily the most compressive set of clauses for the given example set. PROGOL can learn ranges and functions with numeric data (integer and floating point) by making use of the built-in predicates "is", <, =<, etc.

The hypothesis language of PROGOL is restricted by the means of mode declarations provided by the user. The mode declarations specify the atoms to be used as head literals or body literals in hypothesis clauses. For each atom, the mode declaration indicates the argument types, and whether an argument is to be instantiated with an input variable, an output variable, or a constant. Furthermore, the mode declaration bounds the number of alternative solutions for instantiating the atom. The types are defined in the background knowledge by unary predicates, or by Prolog built-in functions.

Arbitrary Prolog programs are allowed as background knowledge. Besides the background theory provided by the user, standard primitive predicates are built into PROGOL and are available as background knowledge. Positive examples are represented as arbitrary definite clauses. Negative examples and integrity constraints are represented as headless Horn clauses. Using negation by failure (CWA), PROGOL is able to learn arbitrary integrity constraints.

PROGOL provides a range of parameters for controlling the generalisation process. These parameters specify the maximum cardinality of hypothesis clauses, a depth bound for the theorem prover, the maximum layers of new variables, and an upper bound on the nodes to be explored when searching for a consistent clause. PROGOL allows to relax consistency by setting an upper bound on the number of negatives that can be covered by an acceptable clause.

3.5 An overview of LINUS

LINUS [LD94, LDG91] is an ILP learner which incorporates existing attribute-value learning systems. The following description was taken from [LWZ$^+$96].

The idea is to transform a restricted class of ILP problems into propositional form and solve the transformed learning problem with an attribute-value learning algorithm. The propositional learning result is then re-transformed into the first-order language. On the one hand, this approach enhances the propositional learners with the use of background knowledge and the more expressive hypothesis language. On the other hand, it enables the application of successful propositional learners in a first-order framework. As various propositional learners can be integrated and accessed via LINUS, LINUS also qualifies as an ILP toolkit offering several learning algorithms with their specific strengths. The present distribution of LINUS provides interfaces to the attribute-value learners ASSISTANT, NEWGEM, and CN2. Other propositional learners may be added. LINUS can be run in two modes. Running in CLASS mode, it corresponds to an enhanced attribute-value learner. In RELATION mode, LINUS behaves as an ILP system.

The basic principle of the transformation from first-order into propositional form is that all body literals which may possibly appear in a hypothesis clause (in the first-order formalism) are determined, thereby taking into account variable types. Each of these body literals corresponds to a boolean attribute in the propositional formalism. For each given example, its argument values are substituted for the variables of the body literal. Since all variables in the body literals are required to occur also as head variables in a hypothesis clause, the substitution yields a ground fact. If it is a true fact, the corresponding propositional attribute value of the example is true, and false otherwise. The learning results generated by the propositional learning algorithms are retransformed in the obvious way. The induced hypotheses are compressed in a postprocessing step.

In order to enable the transformation into propositional logic and vice versa, some restrictions on the hypothesis language and background knowledge are necessary. As in most systems, training examples are ground facts. These may contain structured, but nonrecursive terms. Negative examples can be stated explicitly or generated by LINUS according to the CWA. LINUS offers several options for controlling the generation of negative examples.

The hypothesis language of LINUS is restricted to constrained deductive hierarchical database clauses, that is, to typed program clauses with nonrecursive predicate definitions and nonrecursive types where the body variables are a subset of the head variables. Besides utility functions and predicates, hypothesis clauses consist of literals unifying two variables (X = Y) and of literals assigning a constant to a variable (X = a). Certain types of literals may appear in negated form in the body of a hypothesis clause.

Background knowledge has the form of deductive database clauses, that is, possibly recursive program clauses with typed variables. The variable type definitions which are required to be nonrecursive have to be provided by the user. The background knowledge consists of two types of predicate definitions, namely utility functions and utility predicates. Utility functions are predicates which compute a unique output value for given input values. The user has to declare their input/output mode. When occuring in an induced clause, the output arguments are bound to constants. Utility predicates are boolean functions with input arguments only. For a given input, these predicates compute true or false.

3.6 An overview of Aleph

Aleph [Sri] stands for A Learning Engine for Proposing Hypothesis. The following description was taken from [Sri].

Aleph is intended to be a prototype for exploring ideas. Earlier incarnations (under the name P-Progol) originated in 1993 as part of a fun project undertaken by Ashwin Srinivasan and Rui Camacho at Oxford University. The main purpose was to understand ideas of inverse entailment [Mug95]. Since then, the implementation has evolved to emulate some of the functionality of several other ILP systems. Some of these of relevance to Aleph are: CProgol, FOIL, FORS, Indlog, MIDOS, SRT, Tilde, and WARMR.

During routine use, Aleph follows a very simple procedure that can be described in 4 steps:

1. **Select example**. Select an example to be generalised. If none exist, stop, otherwise proceed to the next step.

2. **Build most-specific-clause**. Construct the most specific clause that entails the example selected, and is within language restrictions provided. This is usually a definite clause with many literals, and is called the “bottom clause”. This step is sometimes called the “saturation” step.

3. **Search**. Find a clause more general than the bottom clause. This is done by searching for some subset of the literals in the bottom clause that has the “best” score. Two points should be noted. First, confining the search to subsets of the bottom clause does not produce all the clauses more general than it, but is good enough for this thumbnail sketch. Second, the exact nature of the score of a clause is not really important here. This step is sometimes called the “reduction” step.

4. **Remove redundant**. The clause with the best score is added to the current theory, and all examples made redundant are removed. This step is sometimes called the "cover removal" step. Note here that the best clause may make clauses other than the examples redundant. Again, this is ignored here. Return to Step 1.

A more advanced use of Aleph allows alteration to each of these steps. At the core of Aleph is the "reduction" step, presented above as a simple "subset-selection" algorithm. In fact, within Aleph, this is implemented by a (restricted) branch-and-bound algorithm which allows an intelligent enumeration of acceptable clauses under a range of different conditions.

3.7 An overview of IndLog

IndLog [Cam00, Cam04] is a general purpose Prolog-based Inductive Logic Programming (ILP) system. The following description was taken from [Cam04].

IndLog is theoretically based on the Mode Directed Inverse Entailment and has several distinguishing features that makes it adequate for a wide range of applications. To search efficiently through large hypothesis spaces, IndLog uses original features like lazy evaluation of examples and Language Level Search. IndLog is applicable in numerical domains using the lazy evaluation of literals technique and Model Validation and Model Selection statistical-based techniques. IndLog has a MPI/LAM interface that enables its use in parallel or distributed environments, essential for Multi-relational Data Mining applications. Parallelism may be used in three flavours: splitting of the data among the computation nodes; parallelising the search through the hypothesis space and; using the different computation nodes to do theory-level search. IndLog has been applied successfully to major ILP literature datasets from the Life Sciences, Engineering, Reverse Engineering, Economics, Time-Series modelling to name a few. IndLog was developed with a modular architecture in mind, being easily configured and enhanced by the introduction of new modules.

3.8 An overview of CLAUDIEN

The interactive system CLAUDIEN [RB93, RD96] performs the task of clausal discovery, that is, it searches a given database for hidden regularities. Both the database and the regularities are represented as first-order clausal theories. CLAUDIEN belongs to non-monotonic setting of ILP. CLAUDIEN regards the discovered regularities as an aim of themselves. As CLAUDIEN provides a powerful mechanism for specifying the type of regularities to be detected, CLAUDIEN can be applied for detecting various kinds of regularities in databases, such as integrity constraints in databases, functional dependencies and determinations, or properties of sequences.

The basic principle of CLAUDIEN's discovery algorithm is to subsequently generate the clauses contained in the hypothesis language and check them against the database. The clauses which are found to represent an actual regularity of the data are added to the hypothesis. The algorithm searches the hypothesis space from general to specific, thereby exploiting the subsumption relations among of clauses for pruning the search space. While running, CLAUDIEN successively enlarges the set of discovered regularities. The longer the algorithm runs, the more regularities may be found. As the search outputs a valid hypothesis whenever it is interrupted, CLAUDIEN can be regarded as an anytime algorithm.

CLAUDIEN's background theory and examples (termed 'observations') are represented as conjunctions of first order Horn clauses. The hypothesis may consist of arbitrary clauses. CLAUDIEN incorporates a mechanism for the syntactical declaration of the hypothesis language called DLab. This mechanism allows the user to specify general clause templates for hypothesis clauses. Each template defines a set of clauses. DLab derives refinement operators from the clause templates which map the expansion of the template into clause sets on sequences of specialisation operations under theta-subsumptions. This enables pruning of the search.

CLAUDIEN provides a range of control parameters. Some of these allow further control of the hypothesis language. Another group concerns semantical aspects of the hypothesis as, e.g., the minimum accuracy and coverage of the discovered clauses. Additionally, the user can choose among four search strategies. Optionally, the system can be requested to produce non-redundant hypothesis, that is, hypothesis not containing clauses which are logically entailed by the background knowledge or other discovered regularities. Furthermore, CLAUDIEN provides mechanisms for the convenient management of different configurations of discovery experiments.

3.9 An overview of TILDE

In the attribute-value learning domain, two major paradigms exist. There are decision tree learners, which follow a divide-and-conquer approach; and rule induction systems, typically following a covering approach. Because of these different search strategies, a rule set derived from a decision tree will usually look different from a rule set found directly by means of a typical rule induction system.

In the ILP domain, up till now most systems have used the covering approach, although some authors (e.g., Bostrom 1995) have already pointed out that the divide-and-conquer strategy can have advantages in some cases.

Recently, an algorithm has been developed at the K.U.Leuven that learns a predicate logic theory by means of so-called logical decision trees. Logical decision trees are a first-order logic upgrade of the classical decision trees used by propositional learners. In the same manner as propositional rules can be derived from decision trees (each rule corresponds to a path from the root to some leaf; the tests in the nodes on that path are conditions of the rule), clauses can be derived from logical decision trees (each test on the path from root to leaf now being a literal or conjunction of literals that is part of the clause). The resulting trees can directly be used for classification of unseen examples, but they can also easily be transformed into a logic or Prolog program.

The ILP setting used by this algorithm, is the "learning from interpretations" setting (De Raedt and Dzeroski 1994), as also used by the CLAUDIEN (De Raedt and Dehaspe 1996) and ICL (De Raedt and Van Laer 1995) systems.

The TILDE [BR98] system is a prototype implementation of this algorithm. It incorporates many features of Quinlan's C4.5, which is a state-of-the-art decision tree learner for attribute-value problems. Next to these, a number of techniques are used that are specific to ILP: a language bias can be specified (types and modes of predicates), a form of lookahead is incorporated, and dynamic generation of literals (DGL) is possible. The latter, based on Claudien's call handling, is a technique that allows, among other things, to fill in constants in a literal. For learning in numerical domains, a discretization procedure is available that can be used by the DGL procedure to find interesting constants.

This implementation, not surprisingly, performs as well as C4.5 on propositional problems (except for lower speed), but experiments on typical ILP data sets also show promising results with respect to predictive accuracy, efficiency and theory complexity.

3.10 An overview of MIS

The Model Inference System, by Ehud Y. Shapiro [Sha91, Sha05] is an instance of the class of incremental generalise/specialise algorithms. Shapiro formalises the Model Inference Problem as follows [Sha91]: Suppose we are given a first order language L and two subsets of it: an obervational language L_o and a hypothesis language L_h with $\Box \in L_o \subset L_h \subset L$. In addition, assume that we are given an oracle for some unknown model M of L. The Model Inference Problem is to find a finite L_o-complete axiomatization of M. Unpacking this definition, we have the following:

- MIS takes as input a sequence of observations, where every statement α in L_o (eventually) appears and is flagged as either *true* or *false*. Such pairs are called *facts*.
- It is assumed that there is an *oracle* that can determine the truth of any ground statement in L relative to the model M. This kind of learning is termed *supervised* as it relies on the existence of an entity capable of answering questions about the

domain. The oracle can be formalised as a set facts $< \alpha, Boolean >$, each of which reflect the truth or falsity of α in the target model M.

- The output of this algorithm is a conjecture, which is a set of Horn clauses that entail all the positive observations, and none of the negatives, that have been presented to it so far.

MIS itself consists of two parts: roughly, a systematic way of finding bugs in programs and a search for fixes for these bugs. Both of these exploit the clean semantics of the pure Horn Clause core of Prolog, and hence run into difficulties with the non-logical parts of the language. MIS was the first system to introduce the notion of refinement operators.

Chapter 4

The BET system

This chapter introduces the BET system, from the modules it provides to the way they are handled and how the system is configured.

4.1 Requirements

The BET system aims to provide a flexible interface to develop new ILP systems and to test new emerging techniques in a quick and straightforward way. The main requirements such a system should consider are:

- Provide a basic set of modules to assemble an ILP system.
- Provide ways of adding new modules to the system.
- Provide ways of structuring the modules in a layered architecture.
- Provide well-documented APIs for developers to have quick access to the available predicates.
- Manage dependencies between modules.
- Manage incompatibilities between modules.
- Provide a configuration format to specify a module.
- Provide a configuration format to specify a system from a subset of the available modules.

4.2 Modular architecture

The BET system is based on a modular architecture which enables different levels of abstraction for the modules provided to synthesise a system. Five levels of abstraction were defined: META level, ILP system's level, ILP intermediate level, ILP basics level and C/Java programming level. A picture of the BET architecture is presented in Figure 4.1.

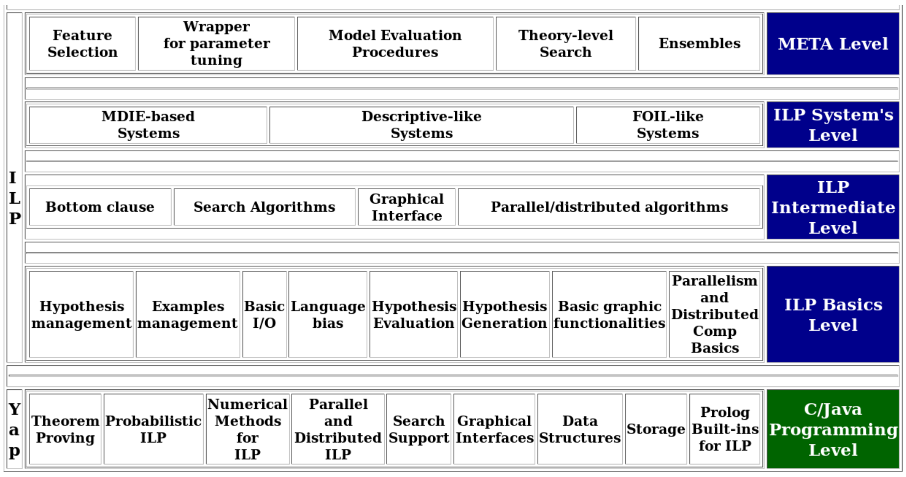


Figure 4.1: Level-based architecture of the Bet system.

The lowest layer (C/Java programming level) provides a set of modules that interface useful libraries or implement in C or Java useful basic functionalities. Currently, we are using the Yap Prolog engine. Theorem proving modules implement depth and time-bound theorem prover and probabilistic theorem prover. For probabilistic ILP, CLPBN is available. The available interface with the R-library provides a large number of numerical methods. Parallelism and distributed computing may use the available interface with the LAM/MPI library. Graphical interfaces use either GTK interface or the Java interface. General purpose data structures include tries, RL-trees, bit-vectors and (ordered) lists. Storage facilities include access data in XML interface for RDBMs (MySQL, ODBC). Prolog built-ins for ILP include tabling support and query packs support.

The "ILP basics level" layer include sets of modules encoding the most basic functionalities of ILP systems. These functionalities include: hypothesis generation (lgg, refinement operators, V and W operators, tools for the random search of hypotheses), management (clause and theory data structures, manipulation sets of clauses, manipulation sets of theories) and evaluation (clause and theory evaluation, clause evaluation on a regression setting, lazy evaluation, stochastic matching, coverage caching, query packs, tabling, parallel coverage tests, intensional and extensional coverage, evaluation metrics) procedures; primitives for parallel execution; management of examples, etc. The basic graphical set of modules enable the visualisation of the search space, the literal dependency graph (when the bottom clause is generated), debugging and profiling. The basic I/O modules include reading from file, reading from DB, post-processing of background knowledge data. Most of the modules in this layer are wrappers and interfaces from the lowest layer code (in C or Java) and ILP intermediate level layer modules.

The ILP intermediate level layer provides functionalities such as search algorithms (systematic search, stochastic search, beam search, incremental searches), graphical user interfaces, the construction of the bottom clause for MDIE-based algorithms and also a set of parallel search algorithms. These modules build on top of modules of the ILP basics level layer.

The ILP system's level layer has a set of modules that, together with modules of lower layers, implement existing state-of-the-art systems or ready to use new systems. For a first version of the BET workbench, we have distinguished between Mode Directed Inverse Entailment (MDIE), declarative ILP systems and FOIL-like systems.

The upper layer (Meta level) provides a set of modules that may facilitate and improve the usage of basic ILP systems. They enable the automatic tuning of parameters of basic ILP systems in the set of modules called "Wrapper for parameter tuning". This facility avoids the user to know details of workings of ILP systems. The set of modules "Theory-level search" provides search strategies to look for the "best" theory produced with the available data. The "Ensembles" set of modules enables the combination of different basic ILP systems. The "Model evaluation procedures" provides procedures such as cross validation, train/test that are useful for the evaluation of the induced theories. Finally the "Feature selection" set of modules provides procedures that select the best subset of background predicates.

4.2.1 C/Java programming level

The C/Java programming level provides a set of basic modules needed for the implementation of an ILP system. This modules can be provided as interfaces for C or Java code, or in Prolog itself. A huge amount of predicates provided here are needed for the implementation of modules of higher levels.

4.2.1.1 Theorem proving

The theorem proving modules are based on mechanisms provided by the Prolog [Bra00] engine itself. The proving of a theorem, based on Horn clauses, can be done by direct application of the SLD resolution [Kow79], which is the basic inference rule used in logic programming, and the one applied in the Yap prolog engine [CDRA]. It is, however, possible for an ILP system to induce an infinite theory, in which there is no end to the recursion. It is therefore important to provide a bound on the proving mechanism, which can be established both by depth and by time constraints.

The Yap Prolog engine already provides predicates for calling a given goal with a given depth of calls. This is easily done in prolog, following its recursive nature, as each call can be accounted for and the status of the stack can be verified at each instance. For time-bound calls, the alarm features Yap provides are used, and an exception is thrown whenever the time has surpassed the given value.

4.2.1.2 Probabilistic ILP

Probabilistic inductive logic programming [RK04, RK03], sometimes also called statistical relational learning, addresses one of the central questions of artificial intelligence: the integration of probabilistic reasoning with first order logic representations and machine learning. A rich variety of different formalisms and learning techniques have been developed.

On the probabilistic ILP setting, the logic programming representations suffers two essential changes:

1. clauses are annotated with probability values

2. the covers relation becomes a probabilistic one

Therefore, a probabilistic covers relation takes as arguments an example e, a hypothesis H and possibly the background knowledge theory B. It then returns a probability value between 0 and 1. So, $covers(e,H \cup B) = P(e|H,B)$, the likelihood of example e. The task of probabilistic ILP then becomes finding the hypothesis H^* that maximises the likelihood of the data $P(E|H^*,B)$, where E denotes the set of examples.

To support probabilistic ILP, the BET system provides access to the CLP(BN) [CC03] module from the Yap prolog engine. This module uses a Bayesian network to represent the joint probability distribution over terms constructed from the Skolem functors in a logic program, with an extension based on constraint logic programming (CLP). This extension enables prolog clauses to have attached probability distributions. In order to use CLP(BN) within an ILP environment, predicates for clause simplification, until no cycles remain, are provided.

4.2.1.3 Numerical methods for ILP

Prolog is known for its limitations when it comes to numerical methods. Since an ILP problem can involve some statistical or numerical calculus, an interface with the R-project [rpr] has been developed. Such an interface enables the calling of methods from the R shared library directly from Prolog predicates. The system is flexible enough for each method in R to be called from a Prolog string, with a list of arguments. Some binary operations, which have a different calling mechanism within R are specified as such in Prolog facts, to enable the correct differentiation when addressing the goal.

4.2.1.4 Parallel and distributed ILP

LAM/MPI [BDV94, SL03] is one of the predecessors of the Open MPI project. Open MPI represents a community-driven, next generation implementation of a Message Passing Interface (MPI) fundamentally designed upon a component architecture to make an extremely powerful platform for high-performance computing.

LAM (Local Area Multicomputer) is an MPI programming environment and development system for heterogeneous computers on a network. With LAM/MPI, a dedicated computer cluster or an existing network computing infrastructure can act as a single parallel computing resource. LAM/MPI is considered to be “cluster friendly”, in that it offers daemon-based process startup/control as well as fast client-to-client message passing protocols. LAM/MPI can use TCP/IP, shared memory, Myrinet (GM), or Infiniband (mVAPI) for message passing.

LAM features a full implementation of MPI-1 and much of MPI-2. Compliant applications are source code portable between LAM/MPI and any other implementation of MPI. In addition to providing a high-quality implementation of the MPI standard, LAM/MPI offers extensive monitoring capabilities to support debugging. Monitoring happens on two levels. First, LAM/MPI has the hooks to allow a snapshot of process and message status to be taken at any time during an application run. This snapshot includes all aspects of synchronisation plus data type maps/signatures, communicator group membership, and message contents (see the XMPI application on the main LAM web site). On the second level, the MPI library is instrumented to produce a cumulative record of communication, which can be visualised either at runtime or post-mortem.

The BET system provides an interface to the LAM/MPI library directly through a module in the Yap Prolog engine.

4.2.1.5 Search support

The search support modules provide support for the search algorithms described in section 4.2.3.2. To enable a generalisation of the search methods, a generic branch-and-bound [LD10] procedure is provided, which allows different implementations of search procedures. Therefore, each node on the search graph (which corresponds to a clause) has a dual search key (primary and secondary). The keys are both the length (i.e. the number of literals) of the clause and the value of the clause given by the evaluation function. The branching is done by applying a refinement operator to the clause represented by the node. The algorithm for such a generic branch-and-bound approach can be seen in algorithm 2.

Algorithm 2 An example of a generic branch and bound procedure for learning rules

```
k = *Number of rules to return*
E = *Set of examples*
Good = ∅
S = START_RULE
**while** stop criteria not satisfied **do**
   $Pick = pickRule(S)$
   $S = S\ \{Pick\}$
   **if** $notprune(Pick)$ **then**
      $NewRules = genNewRules(Pick)$
      $NewRules = \{r \in NewRules | notprune(r)\}$
      $evalOnExamples(E, NewRules)$
      $Good = Good \cup \{r \in NewRules | good\_rule(r)\}$
      $S = S \cup (NewRules\ Good)$
   **end if**
**end while**
return $bestOf(k, Good)$
```

4.2.1.6 Graphical interfaces

ILP is usually hard to be used by people without knowledge in the area. The BET system, by providing a simple workbench for the creation of ILP systems, aims to reduce the overhead necessary for some people to know some ILP basics. By providing graphical interfaces for the introduction of examples, background knowledge, tweaking the system and running the induction procedure, users may abstract themselves from the underlying prolog engine, and use ILP in a more familiar way. The BET system aims to provide graphical interfaces in both GTK and Java Swing.

4.2.1.7 Data structures

There are a lot of interesting data structures for the ILP problem which are provided in the BET system. The BET system aims to provide access to Tries, RL-trees, BIT vectors and (ordered) lists.

Tries were originally invented by Fredkin [Fre60] to index dictionaries and have since been generalised to index recursive data structures such as terms.

The basic idea behind the trie data structure is to partition a set T of terms based upon their structure so that looking up and inserting these terms will be efficiently done. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term [FCR$^+$09].

An essential property of the trie structure is that common prefixes are represented only once. The efficiency and memory consumption of a particular trie data structure largely depends on the percentage of terms in T that have common prefixes. For ILP systems, this is an interesting property that we can take advantage of because the hypothesis space is structured as a lattice and hypotheses close to one another in the lattice have a largely common structure. More specifically, hypotheses in the search space have common prefixes (literals), and some related information is also similar (e.g. the list of variables in an hypothesis is similar to other lists of variables of nearby hypotheses). This clearly matches the common prefix property of tries.

At the entry point of a trie we have the root node. Internal nodes represent symbols in terms and leaf nodes specify the end of terms. Each root-to-leaf path represents a term described by the symbols labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing symbol. Inserting a new term requires traversing the trie starting at the root node. Each child node specifies the next symbol to be inspected in the input term. A transition is taken if the symbol in the input term at a given position matches a symbol on a child node. Otherwise, a new child node representing the current symbol is added and an outgoing transition from the current node is made to point to the new child node. On reaching the last symbol in the input term, we reach a leaf node in the trie. Variables in terms are represented as a distinct constant. A representation of the internal of terms in tries can be seen in Figure 4.2. The trie data structure is implemented in C as a shared library, and provided as a Yap module.

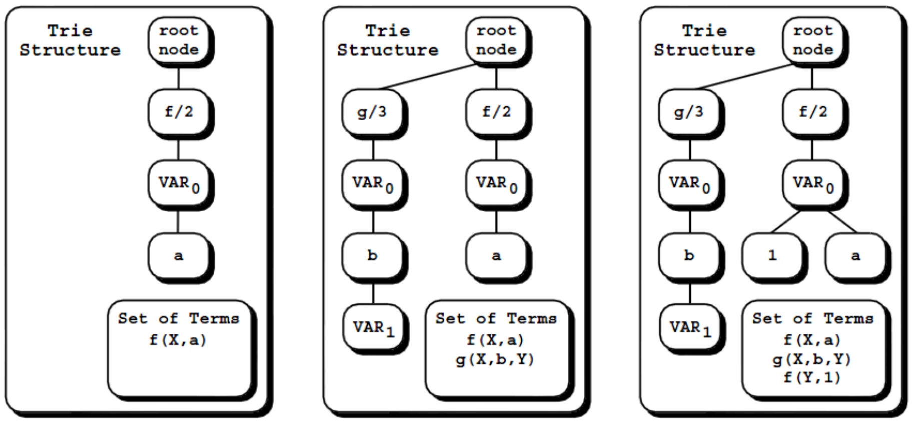


Figure 4.2: Terms represented in tries

The RL-tree (RangeList-Tree) [FRCS03] data structure is based on a generic data structure called quadtree [Sam84] that has been largely used in application areas such as Image Processing, Computer Graphics, or Geographic Information Systems. Quadtree is a term used to represent a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. Quadtrees based data structures are differentiated by the type of data that they represent, the principle guiding the decomposition process, and the number of times the space is decomposed. The RL-tree data structure is designed to store lists of intervals representing integer numbers. For example, the list $[1,2,5,6,7,8,9,10]$ is represented as the list of intervals

$[1-2,5-10]$.

To reduce the time spent on computing clauses coverage some ILP systems maintain lists of examples covered (coverage lists) for each hypothesis that is generated during execution. The data structure used to maintain coverage lists is usually a Prolog lists of integers. Two lists are kept for each clause: the list of positive examples covered (numbered from 1 to $|E^+|$) and the list of negative examples covered (numbered from 1 to $|E^-|$). Each example is represented by a number in the list. The above mentioned systems reduce the size of the coverage lists by transforming a list of numbers into a list of intervals. Using a list of intervals to represent coverage lists is an improvement over lists of numbers but it still presents some problems. First, the efficiency of performing basic operations on the interval list is linear on the number of intervals. Second, the representation of lists in Prolog is not very efficient regarding memory usage. The RL-trees aim to tackle these problems, achieving efficient storage and manipulation of coverage lists.

In the design and implementation of RL-trees, the following assumptions must be taken into account: intervals are disjoint; updates consist of adding or removing numbers; and, the domain (an integer interval) is known at creation time.

RL-trees have two distinct types of nodes: list nodes (represent a fixed interval of size *LI*) and range nodes (represent an interval that is subdivided into sub-intervals). The number of sub-intervals in each range node is *B* (an implementation parameter).

In the vein of quadtrees, each “cell” (square) in a node is painted black if the example is covered (list nodes) or the interval is completely covered (range nodes). It is painted gray if the interval is partially covered (range nodes) and is white if the example is not covered (list nodes) or the interval is not covered at all (range nodes), as can be seen in Figure 4.3.

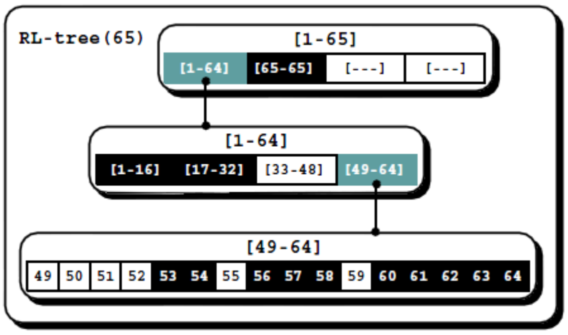


Figure 4.3: Representation of interval $[1-32,53-54,56-58,60-65]$ in a RL-tree(65)

Again, as in quadtrees, the underlying idea of RL-trees is to represent a disjoint set of intervals in a domain by recursively partitioning the domain interval into equal sub-intervals. The number of partitions performed depend on *B*, the size of the domain, and the size of list node interval *LI*. Since we are using RL-trees to represent coverage lists, the domain is $[1,NE]$ (denoted as RL-tree(NE)) where NE is the number of positive or negative examples. The RL-tree data structure is implemented in C as a shared library, and provided as a Yap module.

A BIT vector is an array data structure that compactly stores individual bits (boolean values). It implements a simple set data structure, storing a subset of $1,2,...,n$ which makes it effective at exploiting bit-level parallelism in hardware to perform operations quickly. A typical bit array stores kw bits, where w is the number of bits in the unit of storage, such as a byte or word, and k is some nonnegative integer. If w does not divide the number of bits to be stored, some space is wasted due to internal fragmentation. BIT vectors are widely used in decision rules representation, namely in discrete environments, as each value can be stored in a fixed number of bits.

For the ILP problem, BIT vectors can be used for the representation of coverage lists, specially when the number of examples is small, or even for the representation of clauses. The latter can be used, for instance, in MDIE-based systems, where clauses are built from literals from the bottom clause. A clause can then be encoded as a BIT vector where each bit represents the presence, or absence, of a literal.

Prolog [Bra00] has built-in support for lists, and they play an important role in the implementation of ILP systems. Of particular usefulness is the ability to maintain an ordered list. This is enabled by providing predicates for sorting lists and insert an element into a sorted list. Other useful predicates, such as list concatenation or deletion of elements are also provided.

4.2.1.8 Storage

Most ILP systems read both the background knowledge and examples as prolog programs. Although they need to be in this form for the induction procedure, one can easily specify them in a different format, namely in XML or in a database. The BET system thus provides interfaces for data access in XML and from RDBMs, performing a transformation of the data to prolog programs.

4.2.1.9 Prolog built-ins for ILP

Some Prolog built-ins can be of particular interest to increase the efficiency of ILP systems. Among these built-ins, the BET system provides access both to tabling and query pack support, through the Yap Prolog engine.

Tabling [War95] consists of storing intermediate solutions to a query so that they can be reused during the query execution process. It can be shown that tabling-based computational rules can have better termination properties than SLD-based models, and termination can be guaranteed for all programs with the *bounded term-size property*.

Tabling is about storing and reusing intermediate answers for goals. Whenever a tabled subgoal *S* is called for the first time, an entry for *S* is allocated in the *table space*. This entry will collect all the answers generated for *S*. Repeated calls to *variants* of *S* are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated for *S*, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either *generator nodes*, corresponding to first calls to tabled subgoals, *consumer nodes*, that consume answers from the table space, and *interior nodes*, that are evaluated by standard SLD-resolution.

Space for a subgoal can be reclaimed when the subgoal has been *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. Note that a number of subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*), and therefore can only be completed together. The completion operation is thus performed by the *leader* of the SCC, that is, by the oldest subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when all the subgoals in a SCC have been completely evaluated.

Tabling is supported in the BET system via the YapTab [RSFC00] module. The YapTab design is WAM based [War83], as is the SLG-WAM [SW]. It implements two tabling scheduling strategies, batched and local. As in the original SLG-WAM [SW], it introduces a new data area, the table space; a new set of registers, the freeze registers; an extension of the standard trail, the forward trail; and the four main tabling operations: tabled subgoal call, new answer, answer resolution and completion.

Query packs [BDD$^+$02, BDR$^+$00] are a technique that enable a more efficient evaluation of a group of queries. This technique tries to group queries with similar prefixes, which can be evaluated only once, through the use of cut and once predicates as a pruning procedure. The combination of the advantage of the disjunctive query with the advantage of the individual query with pruning results in the notion of the *query pack*. A query pack can be represented as a tree. This representation enables a better notion of the pruning procedure. When a branch has succeeded, it is effectively pruned away from the pack during the evaluation of the query pack. This pruning is recursive, i.e., when all branches in a subtree of the query pack have succeeded, the whole subtree must be pruned. Evaluation of the query pack then terminates when all subtrees have been pruned or all the remaining queries fail for the example. The use of query packs may enable a more efficient use of background knowledge, in order to improve the efficiency of clause evaluation.

4.2.2 ILP basics level

The ILP basics level layer provides a sets of modules which comprise the most basic functionalities of ILP systems, including hypothesis generation, management, manipulation and evaluation, primitives for parallel execution, management of examples, graphical utilities and basic input-output modules. Most of the modules in this layer are wrappers and interfaces from the lowest layer code and intermediate level modules.

4.2.2.1 Hypothesis management

The modules for hypothesis management allow an efficient way both to store and to handle the generation of new hypothesis. Best hypothesis are asserted in the prolog database, along with a label defining their positive cover, negative cover, length and gain. Clauses can be added to the current theory. The current theory is also asserted in the database, consisting of indexed hypothesis.

The BET system also provides a module for clause manipulation. There are predicates for flattening variables in clauses, extract a given term by index, removal of symmetric literals (for a bottom clause), removal of mode repeats, removal of repeated literals because of commutativity or useless literals (ones that do not contribute to establish variable chains to output variables in the head of a clause).

4.2.2.2 Examples management

The BET system provides two alternatives for example management, one that uses `assert` predicates, and one that uses `recorda` predicates. The main difference lies on the fact that `recorda` uses Yap internal database (i.d.b. for short), which is faster, needs less space and provides a better insulation of program and data.

4.2.2.3 Basic I/O

In terms of reading examples, the BET system provides ways of obtaining them both from a prolog or XML file or from a database, based on the interfaces described in section 4.2.1.8. When reading from a database, the BET system automatically declares predicates with the name of the table in singular form and each field as an argument (i.e. if the table is named `daughters` and has, in a row, values `ann` and `tom`, a predicate `daughter(ann, tom)` is declared).

4.2.2.4 Language bias

Since the ILP problem is potentially infinite, reducing the search space is often necessary. One way of doing that is by introducing bias [Mit80], namely on the language level. The

BET system provides modules to add support for determination [Dav87], mode and type declarations [Mug95].

Determination declarations specify, for each target predicate symbol, which other predicate symbols from *B* can appear in its definition. They take the form of:

```
determination(TargetName/Arity1, BackgroundName/Arity2).
```

The first argument is the name and arity of the target predicate (i.e., the predicate that appears in the head of hypothesised clauses). The second argument is the name and arity of a predicate that can appear in the body of such clauses.

Mode declarations specify the mode of call for predicates that can appear in the hypotheses generated by the system. These declarations specify the arguments' types and if they are intended to be an input or an output argument. There may be more than one mode declaration for each predicate symbol except for the head of the target predicate. Mode declarations take the form `modeh(1, PredicateMode)` for the target predicate, and `modeb(RecallNumber, PredicateMode)` for the the background knowledge. The number of possible outputs, for each combination of input arguments, is limited by the `RecallNumber`. `RecallNumber` can either be a number specifying the number of successful calls to the predicate, or $\star$ meaning that all answers are to be used. It is usually simpler to specify `RecallNumber` as $\star$ with the side effect that the system may become slower.

PredicateMode specifies the arguments mode of a predicate. It has the form:

```
predicatename(ModeType, ModeType...).
```

Each `ModeType` is either simple or structured. A simple `ModeType` is one of the form:

- `+T` specifying that when a literal with predicate symbol `predicatename` appears in a hypothesised clause, the corresponding argument should be an "input" variable of type `T`.
- `-T` specifying that the argument is an "output" variable of type `T`.
- `#T` specifying that it should be a constant of type `T`.

4.2.2.5 Hypothesis evaluation

The hypothesis evaluation module offers options to evaluate hypothesis on different settings, using mainly 4 values of a hypothesis: its positive cover (*P*), its negative cover (*N*), its length (*L*) and its gain (*G*), based on the evaluation function currently being used.

The BET system currently has the following different evaluation functions:

Compression This approach takes into account both the coverage and the size of the current clause:

$$Val = P - N - L + 1 \tag{4.1}$$

Coverage This approach takes into account the difference between the number of positive and negative examples covered:

$$Val = P - N \tag{4.2}$$

Laplace Estimate The Laplace estimate is a probability estimation that says that, in the absence of any evidence (i.e. unknown positive cover), the value should be 0.5. Otherwise, its value should be:

$$Val = \frac{P+1}{P+N+2} \tag{4.3}$$

Weighted Relative Accuracy Clause utility is calculated using the weighted relative accuracy function [PFZ99], which states that:

$$\begin{aligned} WRAcc(H \leftarrow B) &= p(B)(p(H|B) - p(H)) && (4.4) \\ &= p(H \cap B) - p(B)p(H) && (4.5) \end{aligned}$$

Therefore, introducing variable $P1$ as the coverage of B, the value can be calculated as:

$$Val = \frac{P}{P+N} - \frac{P1}{P+N} \times \frac{P+N}{P+N} \tag{4.6}$$

Entropy Clause utility is calculated using the entropy formula:

$$Val = -\frac{(\frac{P}{P+N} \times log(\frac{P}{P+N}) + (1 - \frac{P}{P+N}) \times log(1 - \frac{P}{P+N}))}{log(2)} \tag{4.7}$$

Gini Index Clause utility is calculated based on the Gini index:

$$Val = 2 \times \frac{P}{P+N} \times (1 - \frac{P}{P+N}) \tag{4.8}$$

Accuracy The accuracy measure gives a relative value to the clause coverage:

$$Val = \frac{P}{P+N} \tag{4.9}$$

Pseudo-Bayes Clause utility is the pseudo-Bayes conditional probability of a clause [Cus93].

Positive-only Bayes Estimate Clause utility is calculated using the Bayesian score [Mug97] (being R the number of randomly-generated samples):

$$Val = log(P) + log(R+2) - \frac{L+1}{P} \tag{4.10}$$

Auto M Estimate Clause utility is the m estimate with the value of m automatically set to be the maximum likelihood estimate of the best value of m.

M Estimate Clause utility is its m estimate [DB92]. The value of m is user defined.

On a regression setting, the clause evaluation is automatically set to the standard deviation of values predicted.

Lazy evaluation of examples [Cam03, FCR$^+$09] is a technique that consists in avoiding or postponing the evaluation of each clause against all examples. The BET system contains three variants of lazy evaluation: lazy evaluation of positive examples, lazy evaluation of negative examples and total laziness.

A hypothesis is allowed to cover a small number of negative examples (the noise level) or none. If a clause covers more than the allowed number of negative examples it must be specialised. Lazy evaluation of negatives can be used when we are interested in knowing if a hypothesis covers more than the allowed number of negative examples or not. Testing stops as soon as the number of negative examples covered exceeds the allowed noise level or when there are no more negative examples to be tested. Therefore, the number of negative examples effectively tested may be very small, since the noise level is quite often very close to zero. If the evaluation function used does not use the negative counting then this produces exactly the same results (clauses and accuracy) as the non-lazy approach but with a reduction on the number of negative examples tested.

One may also allow the positive cover to be evaluated lazily (lazy evaluation of positives). A clause is either specialised (if it covers more positives than the best consistent clause found so far) or justifiably pruned away otherwise. When using lazy evaluation of positives it is only relevant to determine if a hypothesis covers more positives or not than the current best consistent hypothesis. We might then just evaluate the positive examples until we exceed the best cover so far. If the best cover is exceeded we retain the hypothesis (either accept it as final if it is consistent or refine it otherwise) or we may justifiably discard it. We need to evaluate its exact positive cover only when accepting a consistent hypothesis. In the event of this latter case we don't need to restart the positive coverage computation from scratch, we may simply continue the test in the point where we left it before.

Lazy evaluation can be taken to the extreme and simply do not evaluate the positive cover (total laziness). The evaluation of a hypothesis is divided in two steps. In the first step, we perform a lazy evaluation of negatives examples. If the clause is inconsistent then we are done, no extra evaluation effort is required and the clause is retained for specialisation. On the other hand, if we find a consistent clause then an exact positive coverage is carried out. The advantage of the total laziness is that for each clause we only test it on the negatives until it covers at least the noise level. One should note that in systems that constraint the number of hypotheses generated, it is necessary to relax the nodes limit constraint (i.e., increase the upper bound limit on the number of generated hypotheses). Although we may generate more hypotheses, we may still gain by the increase in speed of their evaluation process since the computational cost of generating hypotheses is usually much inferior than the cost of evaluating them.

Stochastic matching [SR97] allows a reduction on the number of matches between examples and candidate hypothesis. Instead of exhaustively exploring the set of matchings between any example and any short candidate hypothesis, one stochastically explores the set of matchings between any example and any candidate hypothesis. The user sets the number of matching samples to consider and thereby controls the cost of induction and classification.

A method to speedup hypothesis evaluation is the storage of results in order to later reduce the number of examples matched against an hypothesis. This can be done in top-down ILP systems by keeping the set of examples (usually termed coverage lists) explained by a hypothesis so that the refinements are only matched against the set of examples that succeeded (matched) the parent hypothesis. A hypothesis c_s is generated by applying a refinement operator to another hypothesis c_g. Let $cover(c_g) = \{e \in E | B \wedge c_g \models e\}$, where c_g is a clause, B is the background knowledge, and E is the set of positive (E^+) and negative examples (E^-). Since c_g is more general than c_s then $cover(c_s) \subseteq cover(c_g)$. Taking this into account, when testing the coverage of c_s it is only necessary to consider examples of $cover(c_g)$, thus reducing the coverage computation time. Cussens extended this scheme by proposing a kind of coverage caching. The coverage lists are permanently stored and reused whenever necessary, hence avoiding the need to recompute the coverage of equivalent clauses. Coverage lists reduce the effort in coverage computation at the cost of significantly increasing memory consumption. Therefore, efficient data structures should be used to represent coverage lists to minimise memory consumption.

The transformation of hypothesis is also useful for query optimisations [CSC+02]. Methods such as the theta-transformation, the cut-transformation, the once-transformation or the smartcall-transformation are useful to reduce the theorem-proving effort of the produced ILP systems.

Tabling is a logic programming technique that performs a kind of dynamic and transparent lazy extensionalization of predicates. It consists in storing intermediate answers for sub-goals so that they can be reused when a repeated call appears thus avoiding redundant re-computation. The results of exploiting tabling in the ILP context showed that it can reduce the execution time at the cost of using large amounts of memory (a problem that needs to be solved to further explore tabling in ILP).

4.2.2.6 Hypothesis generation

Hypothesis generation is an important aspect in every ILP system. The BET system thus provides different ways for the generation of hypothesis that can be used when building a system.

For least-general generalisation based systems, the BET system provides a predicate which takes two clauses and computes their lgg. The least general generalisation operator is defined as follows:

lgg of terms $lgg(t1,t2)$

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 $lgg(s,t)$

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

lgg of clauses $lgg(c_1,c_2)$

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

For top-down systems, refinement operators, namely based on MDIE [Mug95], which take literals from the bottom clause according to the model and type language, are supported.

V and W operators are also to be supported, implementing the following rules of inverse resolution:

Absorption

$$\frac{q \leftarrow A \qquad p \leftarrow A, B}{q \leftarrow A \qquad p \leftarrow q, B} \tag{4.11}$$

Identification

$$\frac{p \leftarrow A, B \qquad p \leftarrow A, q}{q \leftarrow B \qquad p \leftarrow A, q} \tag{4.12}$$

Intra-Construction

$$\frac{p \leftarrow A, B \qquad p \leftarrow A, C}{q \leftarrow B \qquad p \leftarrow A, q \qquad q \leftarrow C} \tag{4.13}$$

Inter-Construction

$$\frac{p \leftarrow A, B \qquad q \leftarrow A, C}{p \leftarrow r, B \qquad r \leftarrow A \qquad q \leftarrow r, C} \tag{4.14}$$

Finally, the BET system provides predicates to randomly generate hypothesis and keep a track of generated hypothesis to avoid repetition.

4.2.2.7 Basic graphic functionalities

The ILP problem is usually computationally expansible and dependent on how both the background knowledge and the examples are defined. Most of the systems currently available are coded in Prolog, which makes it somehow difficult to inspect both the search space or the literal dependency in the produced clauses. Having access to this information could enable the practitioner to have a better control on how he is defining the problem and most likely optimise its definition to enable a faster and better production of hypothesis.

4.2.2.8 Parallelism and distributed computation basics

In terms of parallelism, the BET system provides some generic predicates to handle distributed computing. These predicates are based on the LAM/MPI interface Yap provides and allow a better management of workers for a given task. Using these predicates, one can know the workers which are busy and the ones which are available, enabling the support for data parallel algorithms.

4.2.3 ILP intermediate level

The ILP intermediate level layer provides intermediate functionalities needed by every ILP system and build on top of modules of the ILP basics level layer.

4.2.3.1 Bottom clause

The generation of the bottom clause is an important step on MDIE-based system. The BET system includes a saturation module, which provides predicates for the creation of the bottom-clause. The saturation predicate follows the algorithm 3.

4.2.3.2 Search algorithms

As with many machine learning problems, the ILP problem is approximated to a search problem on the graph of possible clauses. In this case, different approaches can be taken when travelling the graph.

Breadth first search is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm. In the BET system, a breadth first search considers and expands clauses based on their size (i.e. number of literals).

Depth first search is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In the BET system, the open list of nodes is transformed into a stack, being the top node always expanded and the result of the expansion also added to the top of the stack. Iterative depth-first search is a general strategy often use in combination with depth-first search, that finds the best depth limit. It does this by increasing the depth limit until a solution is found. The main drawback of *ids* is the waste of computations.

Best first search is an informed search which explores a graph by expanding the most promising node each time. The selection of the most promising node in the BET system is based on the evaluation function used, which then becomes the primary key in the dual key system described in section .

Local search [RN02] is a meta-heuristic for solving computationally hard optimisation problems. Local search can be used on problems that can be formulated as finding a solution maximising a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) until a solution deemed optimal is found or a time bound is elapsed. Local search

Algorithm 3 The saturation procedure for MDIE-based systems

```
$k = 0$
$hash : Terms \rightarrow N$
$\bar{e} = \bar{a} \wedge b_1 \wedge .. \wedge b_n$
$-_i = \langle\rangle$
$InTerms = \emptyset$
$m = modeh\ in\ M\ such\ that\ a(m) \preceq a\ with\ substitution\ \theta_h$
if $m = \Box$ then
   return $\Box$
end if
$a_h = a(m)$
for all $v/t$ in $\theta_h$ do
   if $v$ corresponds to a #$type$ in $m$ then
      replace $v$ in $a_h$ by $t$
   else
      replace $v$ in $a_h$ by $v_k$ where $k = hash(t)$
      if $v$ corresponds to $+type$ then
         $InTerms = InTerms \cup \{v\}$
      end if
   end if
end for
$-_i = -_i \cup \{a_h\}$
loop
   if k = i then
      return $-_i$
   end if
   $k = k + 1$
   for all modeb $m$ in $M$ do
      $V = variablesof + typeina(m)$
      $T(m) = set\ of\ n\text{-}tuples\ of\ terms\ such\ that\ each\ T_i\ corresponds\ to\ the\ set\ of\ all\ terms\ of\ the$
      $type\ associated\ with\ v_i\ in\ m$
      for all $\langle t_1, .., t_n \rangle$ in $T_m$ do
         $a_b = a(m)$
         $\theta = \{v1/t1, .., v_n/t_n\}$
      end for
      if goal $a_b\theta$ succeeds with the set of answer substitutions $\Theta_b$ then
         for all $\theta_b$ in $\Theta_b$ do
            for all $v/t$ in $\theta_b$ do
               if $v$ corresponds to a #$type$ in $m$ then
                  replace $v$ in $a_b$ by $t$
               else
                  replace $v$ in $a_b$ by $v_k$ where $k = hash(t)$
                  if $v$ corresponds to $-type$ then
                     $-_i = -_i \cup \{\bar{a}_b\}$
                  end if
               end if
            end for
         end for
      end if
   end for
end loop
```

is possible whenever there is a definition of neighbourhood relation on the search space. For the ILP problem, the lattice structure defines such a neighbourhood relation and thus enables the application of local search on this problem.

Stochastic clause selection [Sri99] restricts the search space by sacrificing optimality. It consists of randomly selecting a fixed-size sample of clauses from the search space with high probability of containing a good clause. This scheme has shown to perform well in the Aleph system even when compared to complex search methods. The quality of the solutions were comparable to the ones found by other search methods, but the time taken to find them was considerable lower.

Algorithm 4 High level description of the Stochastic Clause Selection algorithm

```
$nr$ = maximum number of rules to return
$E$ = set of examples
$C$ = constraints
$k$
$\alpha$ = probability of a rule to be on the top 100 x $k$ percentile
$|\hat{L}|$ = estimate of the number of elements in $L$
$n = \frac{ln(1-\alpha)}{ln(1-k)}$
$N = \{r_1, ..., r_n\}$ be $n$ numbers randomly selected without replacement from 1 to $|\hat{L}|$
$Clauses = \{c_1, ..., c_n\}$ where each $c_i \in Clauses$ is obtained by mapping the number $r_i$ into a clause in $L$
$evalOnExamples(E, Clauses)$
return $bestOf(nr, Clauses)$
```

The Randomised Rapid Restarts (RRR) [ZSP02] search algorithm performs an exhaustive search up to a certain point and then, if a solution is not found, restarts at a different part of the search space. With this approach the search algorithm may avoid being trapped in a very costly path and exploits the high chance of obtaining a better path. The application of the RRR in two applications yielded a drastic reduction of the search times with the cost of just a small loss in predictive accuracy.

4.2.3.3 Graphical interface

The graphical interface modules on ILP's intermediate level interact with interfaces provided by the lower levels in order to provide a better experience on the usage of an ILP system, namely for a non-practitioner of the area.

4.2.3.4 Parallel/distributed algorithms

The BET system provides parallelisation techniques in order to improve performance on the areas of parallel search and parallel matching. All algorithms follow a master-workers approach, in which there is a processor which controls the operations (the *master*) and a pool of workers able to do some task.

Algorithm 5 High level description of the Randomised Rapid Restarts algorithm

```
$E$ = set of examples
$C$ = constraints
$maxtries$ = upper bound on the number of rapid searches performed
$maxtime$ = upper bound on the time that a rapid search may take
$tries = 1$
while $tries \leq maxtries$ do
  select a random clause $c_0$ from $L$
  $searchtime = 0$
  while $searchtime < maxtime$ and an acceptable clause $c$ is not found do
    perform exhaustive radial search starting at $c_0$
  end while
  if $c$ was found then
    return $\{c\}$
  end if
  $tries = tries + 1$
end while
return $\emptyset$
```

The search for a hypothesis involves traversing a search space in some way (e.g., top-down, bottom-up, bidirectional). The strategy of exploring the search space in parallel involves some division of the search space among the processors. Then each processor explores, in parallel, its part of the search space to find a suitable hypothesis. The degree of parallelism and granularity of this strategy depends on the approach adopted to divide the search space.

The Parallel Stochastic Clause Selection (PSCS) [Fon06] algorithm is a parallelised version of Srinivasan's Stochastic Clause Selection [Sri99]. The idea of the algorithm is to randomly select a fixed-size sample of clauses from the search space. The sample will contain a clause in the top k percentile with some probability (both defined by the user). The best clause from the sample is selected. In the beginning of the execution the master replicates the data among all workers. The master randomly draws a set of clauses from the search space and then distributes the clauses evenly among the workers. Each worker evaluates the subset of clauses received on the local data and, after evaluating all clauses, sends the best one to the master. The master then receives the best clauses found by each worker. The best rule received is then returned as the result of the learning procedure.

The parallel coverage test strategy consists in performing the coverage test in parallel, i.e., for each example $e \in E$ the coverage test $(B \wedge H \wedge c \vdash e)$ is performed in parallel. The degree of parallelism depends on the number of examples evaluated in parallel by each processor. The granularity is relatively low but can be enlarged either by increasing the number of examples in each processor or/and by evaluating several rules in parallel instead of a single one.

The parallel coverage tests algorithm (PCT) [Fon06] exploits parallelism by dispatching clauses to workers for evaluation on the local subset of examples. The master's algorithm is similar to the covering algorithm of generic ILP systems with three main changes: first, the examples are divided evenly among the processors in the beginning of the execution and are then loaded by each worker; secondly, the evaluation of a rule is split among all the workers containing examples; thirdly, the removal of examples covered is performed in parallel on all workers.

4.2.4 ILP system's level

The ILP system's level layer has a set of modules that, together with modules of lower layers, implement existing state-of-the-art systems or ready to use new systems. Modules on this level generally provide the main induction predicate.

4.2.4.1 MDIE-based systems

Mode-Directed Inverse Entailment has become one of the most important mechanisms in ILP since its inception by Stephen Muggleton. The BET system provides ways of developing MDIE-based systems by providing 2 distinct modules necessary for their creation.

After saturation, the reduction procedure tries to select the clause with maximal compression built from literals from the bottom clause. The bottom clause thus establishes a lower bound on the search lattice. In order to avoid redundancy and to keep the clauses constructed inside the lattice (i.e. $\Box \preceq H \preceq -_i$) a refinement operator is defined, which selects literals from the bottom clause which can be added to the current clause. This choosing, to avoid redundancy, can be done in different ways, depending on how the bottom clause is stored. If, for each literal of the bottom clause, a list of siblings (i.e. dependant literals) is stored, then a literal who is dependant of one already present in the clause can be chosen. In case no list of siblings is stored, then, by keeping an order of the literals of the bottom clause, a literal who appears next on the bottom clause and which respects the mode declarations can be chosen to be added to the current clause.

Having a refinement operator defined, the reduction procedure then proceeds by searching trough the lattice of clauses. This search is dependant on the search method that is defined, as described in section 4.2.3.2. This search is guided by clause evaluation procedures, as described in section 4.2.2.5.

Defining the saturation and reduction procedures for MDIE-based systems, the induction procedure then follows by selecting an example not yet covered, use it as a seed for the generation of a bottom-clause, apply a reduction method for the generation of clauses, select the best generated clause and add it to a pool of clauses, which will later define our induced theory.

4.2.4.2 Descriptive-like systems

Descriptive-like systems aim to discover regularities, uncovering patterns aimed at solving knowledge discovery in databases (KDD) tasks. Instead of relying on hypothesis generation, this kind of systems have a more exploratory data analysis approach.

The induction procedure of descriptive-like systems therefore aim at finding an hypothesis that is optimal with respect to some quality criterion, as can be the predictive accuracy, therefore softening the coverage criteria most systems rely upon. Descriptive ILP then induces a set of (general) clauses.

The BET system thus provides induction mechanisms for learning first-order decision trees, association rules and performing first-order clustering, all tasks suitable for the *non-monotonic setting* of ILP, as implemented in descriptive ILP systems.

4.2.4.3 FOIL-like systems

Foil like systems provide a top-down approach to induction, starting from a clause with an empty body, which is specialised by repeatedly adding a body literal to the clause built so far. Candidate body literals are predicates which are variabilised (including the target predicate). In order to keep a chain between literals, the induction procedure takes into account (in)equality of variables in the head or body of literals. Literals may contain constants, which are declared by the user as theory constants. Predicates define type restrictions the literals should conform to.

For the pruning of search space, these kind of systems provide parameters for limiting the total number and maximum depth of variables in a single clause, mechanisms for excluding literals which might lead to endless loops in recursive hypothesis clauses. The choice of the literal to be added to a clause takes into account the information gain heuristic, an information-based measure estimating the utility of a literal in dividing positive from negative examples.

In a post-processing phase, unnecessary literals and redundant clauses are removed from the theory built.

4.2.5 Meta level

The meta level provides a set of modules that may facilitate and improve the usage of basic ILP systems, although not necessary for their complete usage.

4.2.5.1 Feature selection

A common method for reducing the size of the hypothesis space in propositional learning is by means of using (relevant) feature subset selection (FSS) [JKP94]. It consists in finding a “good” set of features (attributes) under some objective function (e.g., predictive accuracy). The problem of feature selection can be seen as a search problem [Lan94], where each state in the search space specifies a subset of possible features. Each subset of features needs to be evaluated, independently of the search strategy used to traverse the space of feature sets. There are several approaches to solve the problem of feature selection, being the filter method and the wrapper method two of the most well known [JKP94]. Filter methods select relevant attributes before starting the induction process. The wrapper method generate a set of candidate features and run the induction algorithm on the training data (using only the candidate features) to evaluate the accuracy of the resulting model. Obviously, the wrapper approach is computational expensive since it invokes the learning algorithm multiple times. Techniques that rely on some kind of feature selection are partially correct, although the final models may be better than the ones found without feature selection [ABM04].

4.2.5.2 Wrapper for parameter tuning

The wrapper for parameter tuning isolates parameters specific to each module in a single format. This allows users to know which parameters are available for each module and enables the possibility to change them in a controlled environment.

4.2.5.3 Model evaluation procedures

The model evaluation procedures module provides utilities for the evaluation of produced theories. Among this utilities, there are options for the separation of examples in test and training sets and performing cross-validation.

4.2.5.4 Theory-level search

An adequate explanation for a set of examples typically requires several clauses. Most ILP systems attempt to construct such explanations one clause at a time. The procedure is usually an iterative greedy set-covering algorithm that finds the best single clause (one that explains or “covers” most unexplained examples) on each iteration. While this has been shown to work satisfactorily for most problems, it is nevertheless interesting to consider implementations that attempt to search directly at the "theory-level". In other words, elements of the search space are sets of clauses, each of which can be considered a hypothesis for all the examples. A randomised search method is available for theory-level search, which basically starts from a clause set *S* and moves to subsequent sets of clauses. These sets can be a result of adding a clause, deleting a clause, adding a literal or deleting a literal.

4.2.5.5 Ensembles

In statistics and machine learning, ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models [HKP06]. For ILP systems, having an ensemble involves having different techniques producing hypothesis, which are then combined for a single theory. Two ensemble techniques are available: bagging and boosting.

Bootstrap aggregating, often abbreviated as bagging, involves having each model in the ensemble vote with equal weight. In order to promote model variance, bagging trains each model in the ensemble using a randomly-drawn subset of the training set.

Boosting involves incrementally building an ensemble by training each new model instance to emphasise the training instances that previous models mis-classified. In some cases, boosting has been shown to yield better accuracy than bagging, but it also tends to be more likely to over-fit the training data.

4.3 System description

The BET system is divided in two different applications: a graphical user interface to select a set of modules for producing a system configuration file, and a compiler to gather the selected modules and produce a system. Figure 4.4 describes how the modules are organised, in terms of classes.

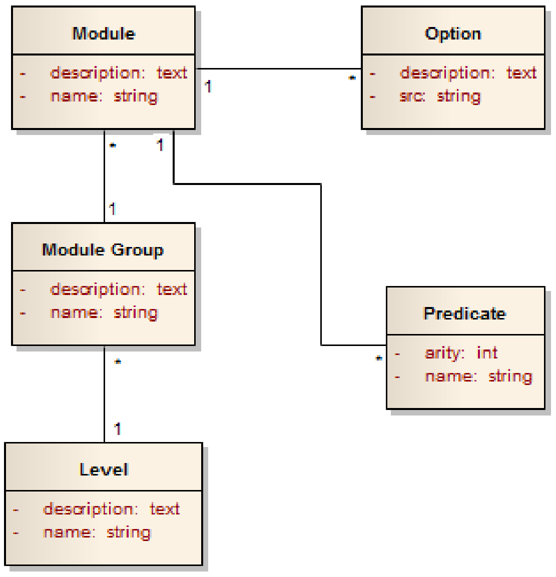


Figure 4.4: Class model of the BET system set of modules

The BET system is therefore organised in 5 different entities:

- **Level**: a level provides a layered group of abstraction for module groups. A module in a given level is typically dependent on modules of the same or lower levels.
- **Module Group**: a module group provides a grouping for a set of modules which are related in the set of functionalities they provide.
- **Module**: a module is the basic entity for the BET system, providing the described functionalities in a set of predicates. A module can have one or more options. A module can have dependent and incompatible modules.
- **Option**: an option is basically an alternative for a module implementation. A module option should provide all predicates the module describes. An ILP system can only have a single module option for a module at a time.
- **Predicate**: A predicate describes how a simple functionality is called within prolog. It has a name and an arity.

In order to provide a flexible way of both describing and adding new modules to the system, an XML-based description was considered. There are 3 different types of XML files: one to describe the system levels, one to describe module groups and one to describe modules.

The BET system accepts a single XML file for the level description, which should be name `levels.xml` and placed under the `conf` directory. A diagram representing the schema for such an XML file can be seen in Figure 4.5. The `config` attribute of the `modulegroup` element should have the path to the XML defining the module group.

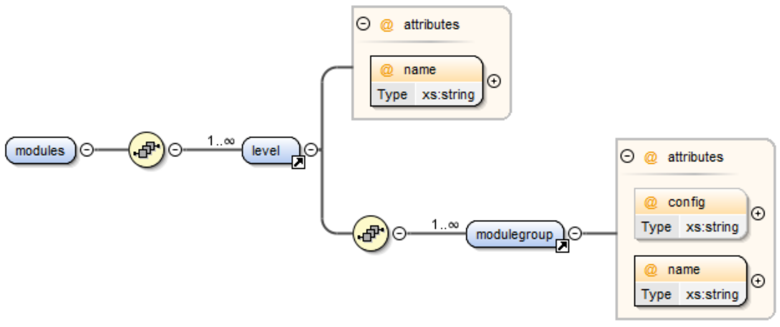


Figure 4.5: Diagram of the XSD schema for the level-defining XML

The module group XML file follows a structure as described in Figure 4.6. The `config` attribute of the `module` element should have the path to the XML defining the module.

The module XML file follows a structure as described in Figure 4.7. The `src` attribute in the `option` element should have the path to the prolog file implementing the

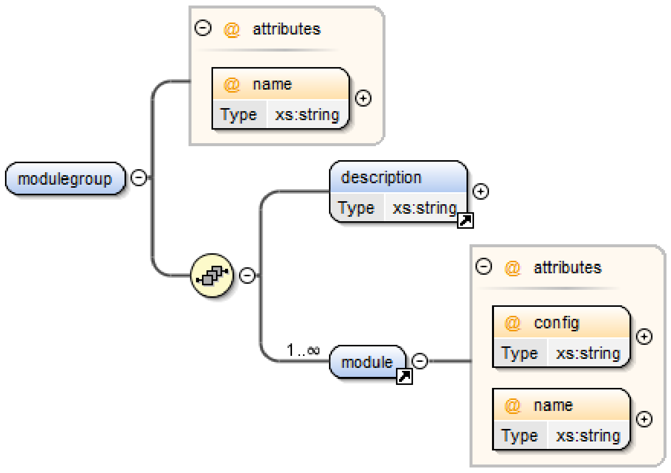


Figure 4.6: Diagram of the XSD schema for the modulegroup-defining XML

module option. The `name` attribute in `incompatibility` or `dependency` elements should include a module name. Since module names are unique, this attribute is enough to univocally identify a module.

Following this approach, whenever an user wants to add a module for the platform, he only needs to define the XML-file for the module description and link it from within the module group files. It is also possible to add new module groups or levels by simple modification of the configuration files. Dependencies and incompatibilities allow the BET system to warn the user whenever the selected system won't build. This initial warning allows better awareness of the user building the system of eventual problems with the final code. If the module has configuration options, these should be included in the module `Wrapper for Parameter Tuning`. Whenever an assert/retract is needed, it should use the `bet_assert` and `bet_retract` predicates on the `General` module.

The GUI interface, loaded with the module configuration as described above can be seen in Figure 4.8.

Besides the graphical user interface, the BET workbench also provides a compiler program. The compiler is executed from the command-line, accepting as parameters a XML file with the system configuration and a destination folder. The compiler warns and halts whenever a dependency or incompatibility isn't respected. In case the system is properly built, a copy of it is created in the destination folder. The system can be run in Yap by consulting the `main.pl` file.

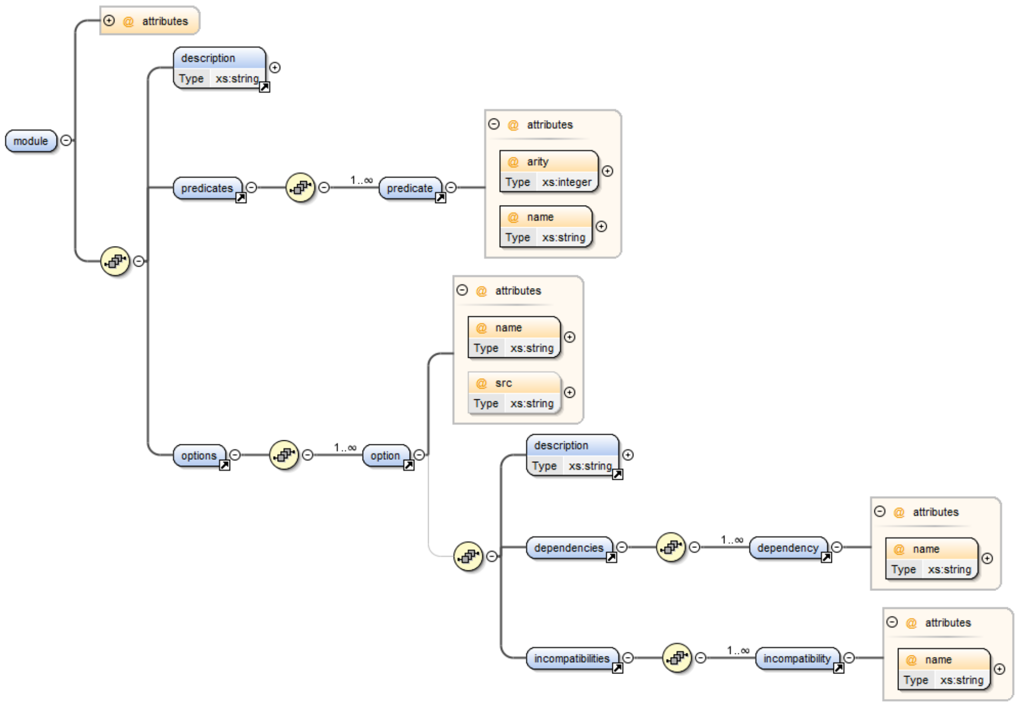


Figure 4.7: Diagram of the XSD schema for the module-defining XML

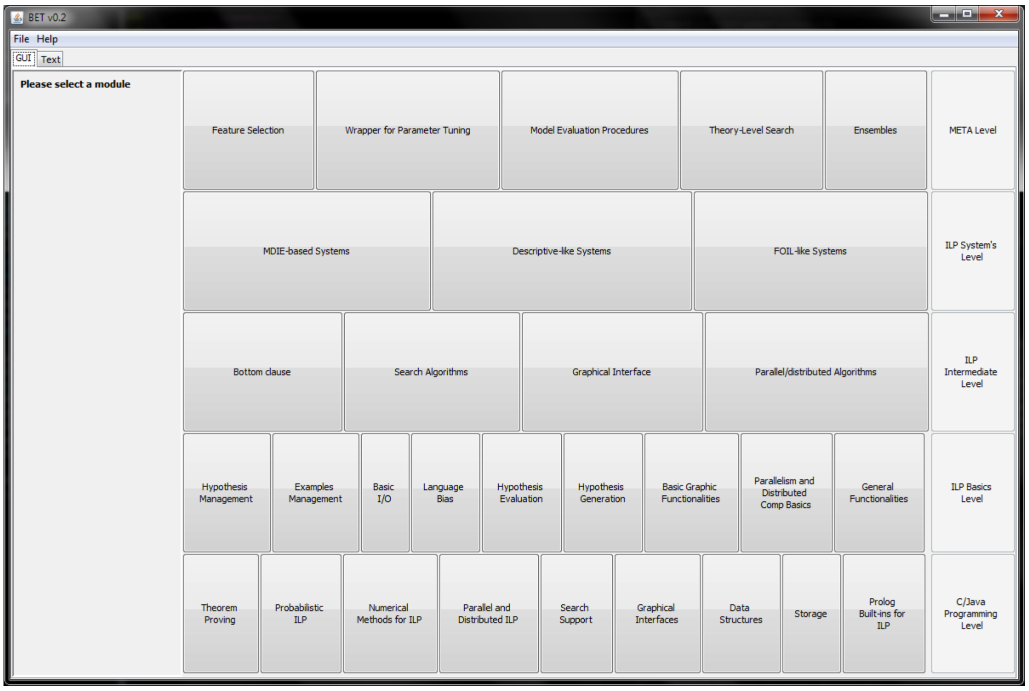


Figure 4.8: The BET workbench graphical user interface for the selection of modules

Chapter 5

Parallelizing search in MDIE-based systems

This chapter introduces a novel technique to explore parallelism in the search space of MDIE-based system. This technique is implemented as a new module for the BET system, providing a way to evaluate the construction and integration of new modules in the framework.

5.1 Reduction step in MDIE-based systems

The reduction step in MDIE-based systems is responsible for the generation and evaluation of potentially good clauses, built from literals from the bottom clause. Progol [Mug95] introduced a refinement operator that is designed both to avoid redundancy and to maintain the relationship $\square \preceq H \preceq \bot$ for each clause H. This build on the notion that, since $H \preceq \bot$, then there is a literal l' in $\bot$ such that $l\theta = l'$. Therefore, there is a uniquely defined subset $\bot\ (H)$ consisting of all l' in $\bot$ for which there exists l in H and $l\theta = l'$. Considering a clause as a subset S' of the set S of literals in the bottom clause, a non-deterministic approach to choosing S' from S involves maintaining an index k. Hence, for each value of k between 1 and n, the cardinality of S, we decide whether to include the kth element of S in S'. The set of all series of n choices corresponds to the set of all subsets of S. Then, for each subset of S there is exactly one series of n choices. To avoid redundancy and maintain the θ-subsumption of $\bot$, both k and θ are kept.

Taking into account mode definitions M, background knowledge B, natural numbers h and i, bottom clause $\bot$ and n as the cardinality of $\bot$, let k be a natural number, $1 \leq k \leq n$. Let C be a clause in $L_i(M)$ and θ be a substitution such that $C\theta \subseteq \bot$. Below a literal l corresponding to a mode m_l in M is denoted simply as $p(v_1, ..., v_m)$ despite the sign of m_l and function symbols in $a(m_l)$. A variable is splittable if it corresponds to a $+$type or $-$type in a modeh of if it corresponds to a $-$type in a modeb. Considering ρ as refinement operators which maps tuples consisting of a clause, a substitution and a natural number to a set of subsequent clauses, one can say that $\langle C', \theta', k' \rangle$ is in $\rho(\langle C, \theta, k \rangle)$ if and only if either

1. $C' = C \cup l$, $k' = k$, $\langle l, \theta' \rangle$ is in $\delta(\theta, k)$ and $C' \in L_i(M)$ or
2. $C' = C$, $k' = k + 1$, $\theta' = \theta$ and $k < n$.

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

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

The variables in $\perp$ form a set of equivalences classes over the variables in any clause C which θ-subsumes $\perp$. The equivalence class of u in θ can be written as the set of all variables in C such that v/u is in θ: $[v]_u$. The second choice in the definition of δ adds a new variable to an equivalence class $[v_j]_{u_j}$. This is refered as *splitting* the variable u_j. A variable is not splittable if it corresponds to +type in a modeb, since the resulting clause would violate the mode declaration language.

5.2 Parallelizing options in MDIE-based systems

There are some known options for parallelizing ILP systems, which can be divided in 4 different areas: parallel exploration of independent hypothesis, parallel exploration of the search space, data parallelism and parallel coverage tests [Fon06]. We're particularly interested in parallel exploration of the search space.

Parallel exploration of the search space implies some division of the search space among the available processors. The degree of parallelism and granularity of the adopted strategy depend on the approach used to divide the search space.

Claudien [RB93, RD96] was one of the first systems supporting parallelization. Its strategy focused on each processor keeping a pool of clauses to specialize, and sharing part of them to idle processors (processors with an empty pool). In the end, the set of clauses found (one set in each processor) are combined and returned as the solution. Claudien, though, follows a non-monotonic setting of ILP. The parallel system, evaulated on a shared-memory computer with two datasets, exhibited a linear speedup up to 16 processors.

Matsui et al. [MISI98] evaluated an algorithm based on, what they called, parallel exploration of the search space. The approach consisted in evaluating, in parallel, the refinements of a clause, correspoding to a strategy based on parallel coverage tests. The results showed very low speedups, according to the authors due to the size of the divided tasks not being the same, hence reducing the efficiency.

Ohwada et. al [ONM00] implemented an algorithm that explores the search space in parallel. The job allocation was dynamic and implemented using contract-net [Smi88] communication. The system showed an almost linear speedup on a ten-processor parallel machine.

Wielemaker [Wie03] implemented a parallel version of the Aleph system for shared memory macines, exploiting parallelism by executing concurrently several randomized local searches [ZSP02]. The system was configured to perform 16 random restarts and made 10 moves per restart, on each processor. Despite low speedups, when compared to other shared memory implementations, the approach could accomplish better results if the granularity of the tasks is enlarged by increasing the number of moves or the number of restarts done by each thread, for example.

Fonseca [Fon06] also proposed some algorithms that exploit parallel exploration of the search space, namely Parallel Stochastic Clause Selection (described in section 4.2.3.4) and Parallel Randomized Random Restarts. Parallel Randomized Rapid Restarts is a parallel version of the Randomized Rapid Restarts algorithm. The PRRR's master algorithm is similar to a sequential covering algorithm, but the master replicates the data to all workers in the beginning of the execution and the removal of examples covered is performed in parallel on all workers. The master always checks for idle workers or blocks until a worker becomes available. Then a random clause is selected from the search space and a message is sent to the worker to start a radial search using the selected clause as a starting point. The result of the search is sent to the master. The master checks if a rule has been received from one of the workers. If found, the loop is interrupted. Then, the master collects all rules found in the other searches that were launched in parallel and the best rule found is selected and returned as the output of the search. PSCS achieves low speedups, since it has a huge communication overhead (a set of clauses is sent to the workers for evaluation) and the time spent in evaluating clauses is low as compared to the time spent randomly generating them. The PRRR shows good speedups with some datasets but has no significant speedups with others.

5.3 Parallelizing search in the reduction step of MDIE-based systems

The proposed algorithm has a master-workers architecture, in order to benefit from the modules the BET system provides to handle such an architecture. The main idea is to benefit from the structure of dependencies of the bottom clause to efficiently divide the search space among the available workers.

The master process is responsible for the saturation procedure and keeping an ordered queue of clauses to be expanded/evaluated. This queue is similar to the openlist of the search mechanisms described in the generic branch and bound approach the BET system provides. By having a similarity to the openlist in search procedures, the master can implement different ways of choosing the clause to expand, let it be depth, breadth or best first search. After saturation, the master then adds the empty clause to the openlist. The workers should all have access to the complete example database and background knowledge.

While it is possible to expand clauses and the master has a non-empty queue, the master sends the top *x* clauses in the queue to the first *x* available workers. It is important that the clauses include information of the last literal of the bottom clause used, in order to prevent the production of repeated clauses. When a worker receives a clause, it proceeds as a regular reduction method in MDIE-based systems, expanding it and evaluating against the example database. Each worker has a parameter, *m*, defining the number of expansions to conduct. Once the number of expansions exceeds that level, the worker sends all resulting expansions to the master and frees itself from work, becoming available for further expansions.

The master is always ready to receive expansions from workers, and adding them to the current queue, possibly using a different strategy as the workers. This enables each worker to have a different search strategy and different ways of traversing the search space. The only drawback of such approach is that randomized algorithms are not supported, as repeated clauses may be generated.

A simplified scheme of the messages exchanged by the proposed algorithm can be seen in 5.1.

5.4 Integrating the module in the workbench

For the purpose of this thesis, we'll describe the ways of integrating the described module on the BET workbench. First of all, there are two alternatives for integrating the piece of code in the framework. Since we define a new reduction procedure, we can either provide this as an option for the reduction module of MDIE-based systems, or as a novel module, defined in a different predicate. We'll be choosing the latter option, as the module defines the new `par_reduce/0` predicate. The first step is defining the XML file for the module, which we'll call `parallel_reduction.xml` and store in the `conf` directory of the BET workbench. The XML looks as follows:

```
<module name="Parallel Reduction">
    <description>
        This module provides a parallel reduction method, based on
        a split of the search space among the processors.
    </description>
    <predicates>
```

Algorithm 6 A high-level description of the proposed algorithm for parallel exploration of the search space

```
E = Set of examples
B = Background knowledge
M = Mode declarations
broadcast(E, B, M)
for all einE do
  $\perp = saturate(e,B,M)$
  broadcast($\perp$)
  loop
    $worker = get\_next\_available\_worker()$
    $R = get\_best\_rule(RulesBag)$
    $send(worker,R)$
    $Results = collect\_data(worker)$
    if $Results = \{R\}$ then
      break
    end if
    $RulesBag = RulesBag \cup Results$
  end loop
  $Rules = get\_best\_rules(RulesBag)$
  $add\_rules\_to\_theory(Rules)$
end for
return $theory$
```

```
        <predicate name="par_reduce" arity="0" />
    </predicates>
    <options>
        <option name="Parallel Reduction"
                src="ilp_system_level/mdie/parallel_reduction.pl" />
    </options>
    <dependencies>
        <dependency name="Saturation" />
        <dependency name="General" />
        <dependency name="Generic utilities for distributed computing" />
        <dependency name="Scheduler/work pool functionalities" />
        <dependency name="Support for data parallel algorithms" />
    </dependencies>
</module>
```

The code for modules should be placed under the `modules` directory. For the sake of organization, the code is also separated for folders corresponding to the levels and module groups defined. In order to link the module in the workbench, we also need to modify the XML corresponding to the module group for MDIE-based systm, to make the system aware of the presence of the module, and the corresponding XML configuration file:

```
<modulegroup name="MDIE-based Systems">
```

```
    <description>
        Mode-Directed Inverse Entailment Systems
    </description>
    <module name="Reduction" config="reduction.xml" />
    <module name="Induction" config="induction.xml" />
    <module name="Parallel Reduction" config="parallel_reduction.xml" />
</modulegroup>
```

The module is then available for usage and compiling in new ILP systems.

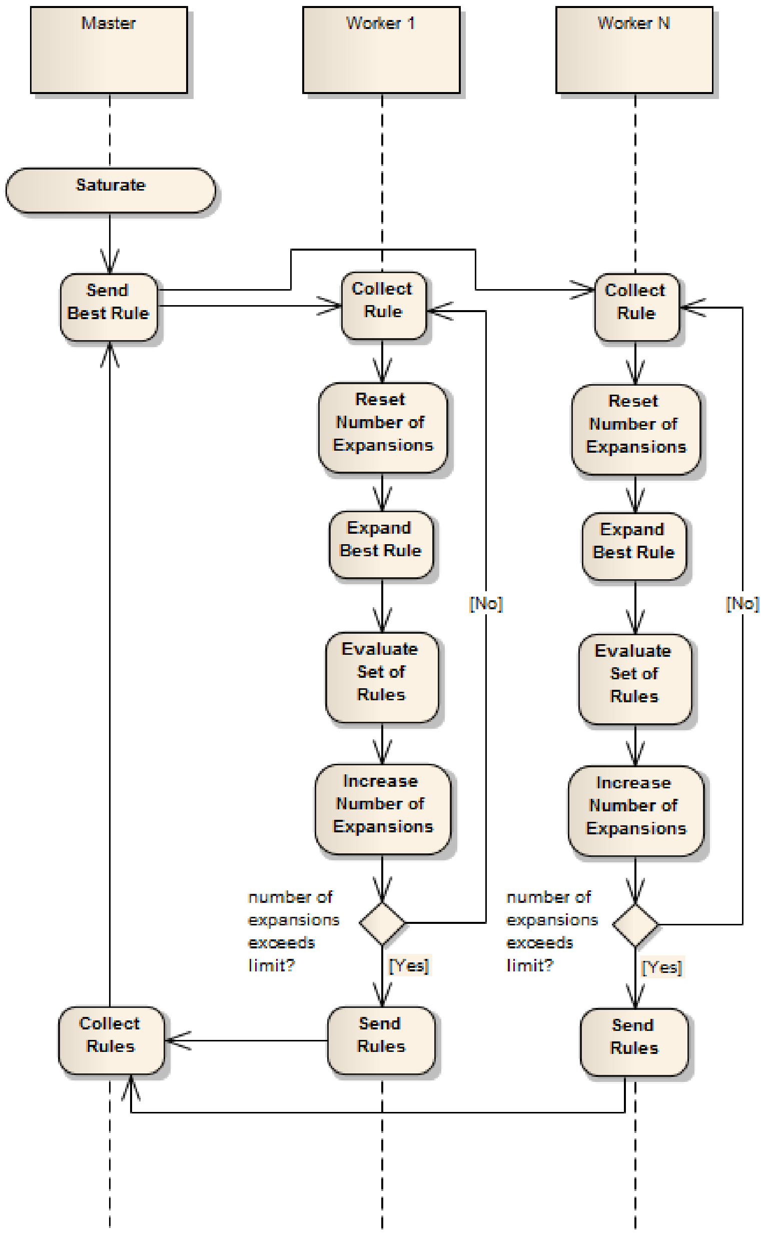


Figure 5.1: Simplified scheme of the messages exchanged by the parallel algorithm

Chapter 6

Conclusions and Future Work

6.1 Conclusions

ILP has been a research area in very active development. By providing both a formal framework and practical algorithms for inductive learning of relational descriptions, it has achieved a great interest for Multi-Relational Data Mining. Its successful applications in both industrial and scientific domains support the notion that ILP is a field with high potential.

One of the main difficulties on the usage and development of ILP systems rely on the need of having some strong theoretical basics on the subject or spend some time understanding a system that implements a given technique. With this work, we aim to provide a way of abstracting the user of the details regarding some techniques.

In this dissertation we studied many different ILP techniques and the ways they relate themselves and developed the BET workbench, a tool capable of quickly synthesizing ILP systems. This tool was created with flexibility in mind, in order to provide easy ways for new modules to be added and for systems to be considered. By relying on XML-configuration files, one can easily modify the tool for his own set of modules.

The advances on the BET workbench should allow a better dissemination of ILP as a research area. By providing a structured collection of interesting modules, we aim to provide both practitioners and curious users with a tool where they can experiment with ILP in a straightforward way. The easier integration of new modules within the workbench should allow new techniques to be tested in a relatively objective way, preventing researchers from worrying with recurrent features every ILP system should consider.

6.2 Future Work

Further work on the BET workbench is still required. Some of the modules described are not yet implemented, or are highly coupled with other modules in the tool. Some predicates are still a bit redundant, as are declared inside different modules, and a clear separation of those would be highly benefitial for the workbench. The tool should also enable the automatic detection of incompatibilities and dependencies from simple code inspection and provide an assisted interface for adding new modules or module options, automatically checking for the presence of the necessary predicates.

The work on the proposed parallelization algorithm is also not yet finished, lacking some tuning on the communication side and real experimental results.

References

[ABM04] Erick Alphonse, Lri Bât, and Stan Matwin. Filtering Multi-Instance Problems to Reduce Dimensionality in Relational Learning. *Journal of Intelligent Information Systems*, 22:23—40, 2004.

[Ari] Aristotle. The Internet Classics Archive | Prior Analytics by Aristotle. http://classics.mit.edu/Aristotle/prior.html.

[Ayc03] John Aycock. A Brief History of Just-in-Time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[Bac09] Francis Bacon. *Novum Organum*. Forgotten Books, September 2009.

[BCR93] Leo Bachmair, Ta Chen, and I. Ramakrishnan. Associative-commutative Discrimination Nets. In *TAPSOFT'93: Theory and Practice of Software Development*, pages 61–74. 1993.

[BDD$^+$02] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, and Henk Vandecasteele. Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. *Journal of Artificial Intelligence Research*, 16:135—166, 2002.

[BDR$^+$00] Hendrik Blockeel, Luc Dehaspe, Jan Ramon, Bart Demoen, Gerda Janssens, and Henk Vandecasteele. Executing Query Packs in ILP. *Proceedings of the 10th International Conference in Inductive Logic Programming, Volume 1866 of Lecture Notes in Artificial Intelligence*, pages 60—77, 2000.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[BFSO84] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Chapman & Hall, 1 edition, 1984.

[BGSS99] M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational Learning: Hard Problems and Phase Transitions. In *Proc. of the 6th Congress AI*IA, LNAI 1792*, pages 178–189, 1999.

[BGSS00] M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational Learning: Hard Problems and Phase Transitions. In *AI*IA 99: Advances in Artificial Intelligence*, pages 178–189. 2000.

[BM91] M. Bain and S. H. Muggleton. Non-monotonic learning. In *Machine Intelligence 12: Towards an Automated Logic of Human Thought*, pages 105–119. Clarendon Press, 1991.

[BMV91] I. Bratko, S. H. Muggleton, and A. Varšek. Learning Qualitative Models of Dynamic Systems. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 385–388, San Mateo, CA, 1991. Morgan Kaufmann.

[BR98] Hendrik Blockeel and Luc De Raedt. Top-down Induction of First-order Logical Decision Trees. *Artificial Intelligence*, 101(1-2):285–297, May 1998.

[Bra00] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, 3rd edition, August 2000.

[Bry90] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *Data Knowledge Engineering*, 5(4):289–312, 1990.

[Bun88] Wray Buntine. Generalized Subsumption and its Applications to Induction and Redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[Cam00] R. Camacho. *Inducing Models of Human Control Skills using Machine Learning Algorithms*. PhD thesis, Department of Electrical Engineering and Computation, Universidade do Porto, 2000.

[Cam02] R. Camacho. Improving the Efficiency of ILP Systems Using an Incremental Language Level Search. In *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.

[Cam03] Rui Camacho. As Lazy as It Can Be. In P. Doherty, B. Tassen, P. Ala-Siuru, and B. Mayoh, editors, *The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03)*, pages 47–58. Bergen, Norway, November 2003.

[Cam04] Rui Camacho. IndLog — Induction in Logic. In *Logics in Artificial Intelligence*, pages 718–721. 2004.

[Car90] Jaime G. Carbonell. Introduction: Paradigms for Machine Learning. In *Machine Learning: Paradigms and Methods*, pages 1–9. Elsevier North-Holland, Inc., 1990.

[CB91] Peter Clark and Robin Boswell. Rule Induction with CN2: Some Recent Improvements. In *Proceedings of the Fifth European Working Session on Learning*, pages 151—163, 1991.

[CC03] Vítor Santos Costa and James Cussens. CLP(BN): constraint logic programming for probabilistic knowledge. *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03)*, pages 517—524, 2003.

[CDRA] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. YAP User's Manual. http://www.dcc.fc.up.pt/ vsc/Yap/.

[CDWY96] Vítor Santos Costa, Costa David, David H. D Warren, and Rong Yang. Andorra-I Compilation. *New Generation Computing*, 14(1):3–30, 1996.

[Ces90] B. Cestnik. Estimating Probabilities: A Crucial Task in Machine Learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 147–149, London, 1990. Pitman.

[Ces91] B. Cestnik. *Estimating Probabilities in Machine Learning*. PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia, 1991.

[CKB87] B. Cestnik, I. Kononenko, and I. Bratko. ASSISTANT 86: A Knowledge-Elicitation Tool for Sophisticated Users. In *Progress in Machine Learning*, pages 31–45. Sigma Press, Wilmslow, UK, May 1987.

[Cla78] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[CN89] Peter Clark and Tim Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3:261—283, 1989.

[CSC00] Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A Note on Two Simple Transformations for Improving the Efficiency of an ILP System. In *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 225–242. Springer-Verlag, 2000.

[CSC$^+$02] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Wim Van Laer, James Cussens, and Alan Frisch. Query Transformations for Improving the Efficiency of ILP Systems. *Journal of Machine Learning Research*, 4:491, 2002.

[CSL07] Vítor Santos Costa, Kostis Sagonas, and Ricardo Lopes. Demand-Driven Indexing of Prolog Clauses. In *Proceedings of the 23rd International Conference on Logic Programming*, pages 305–409. Springer, 2007.

[Cus93] James Cussens. Bayes and Pseudo-Bayes Estimates of Conditional Probabilities and Their Reliability. *Proceedings of the European Conference on Machine Learning*, pages 136—152, 1993.

[Dav87] Todd R Davies. A Logical Approach to Reasoning by Analogy. *IN IJCAI-87*, pages 264—270, 1987.

[DB92] S. Džeroski and I. Bratko. Handling Noise in Inductive Logic Programming. In *Proceedings of the Second International Workshop on Inductive Logic Programming*, Tokyo, Japan, 1992. ICOT TM-1182.

[DD91] S. Džeroski and B. Dolšak. A Comparison of Relation Learning Algorithms on the Problem of Finite Element Mesh Design. In *Proceedings of the XXVI Yugoslav Conference of the Society for ETAN*, Ohrid, Yugoslavia, 1991. In Slovenian.

[DDRW95] S. Džeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of River Water Quality Data Using Machine Learning. In *Proceedings of the 5th International Conference on the Development and Application of Computer Techniques to Environmental Studies*, 1995.

[DL91] S. Džeroski and N. Lavrač. Learning Relations from Noisy Examples: An Empirical Comparison of LINUS and FOIL. In L. Birnbaum and G. Collins, editors, *Proceedings of the 8th International Workshop on Machine Learning*, pages 399–402. Morgan Kaufmann, 1991.

[DL01] Saso Dzeroski and Nada Lavrac. *Relational Data Mining*. Springer, August 2001.

[DM92a] Bojan Dolsak and Stephen Muggleton. The Application of Inductive Logic Programming to Finite Element Mesh Design. *Inductive Logic Programming*, pages 453–472, 1992.

[DM92b] B. Dolšak and S.H. Muggleton. The Application of Inductive Logic Programming to Finite Element Mesh Design. In S. H. Muggleton, editor, *Inductive Logic Programming*, pages 453–472, London, 1992. Academic Press.

[Dol91] B. Dolšak. Constructing Finite Element Meshes Using Artificial Intelligence Methods. Master's thesis, Faculty of Technical Sciences, University of Maribor, Maribor, Slovenia, 1991. In Slovenian.

[DR95] Luc Dehaspe and Luc De Raedt. Parallel Inductive Logic Programming. *Proceedings of the MLNet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 112–117, 1995.

[Dze] S. Dzeroski. ILP Network of Excellence. http://www-ai.ijs.si/ ilpnet2/.

[FCR$^+$09] Nuno A. Fonseca, Vítor Santos Costa, Ricardo Rocha, Rui Camacho, and Fernando Silva. Improving the Efficiency of Inductive Logic Programming Systems. *Software - Practice and Experience*, 39(2):189–219, 2009.

[Fen91] C. Feng. Inducing Temporal Fault Diagnostic Rules from a Qualitative Model. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 403–406, San Mateo, CA, 1991. Morgan Kaufmann.

[Fon06] Nuno A. Fonseca. *Parallelism in Inductive Logic Programming Systems*. PhD thesis, University of Porto, 2006.

[FRCS03] Nuno Fonseca, Ricardo Rocha, Rui Camacho, and Fernando Silva. Efficient data structures for inductive logic programming. In *Inductive Logic Programming*, pages 130–145. 2003.

[Fre60] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, 1960.

[Gö01] Kurt Gödel. *Kurt Gödel: Collected Works*. Oxford University Press, July 2001.

[Gra96] Peter Graf. *Term Indexing*. Springer-Verlag New York, Inc., 1996.

[HKP06] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2 edition, June 2006.

[Hwa89] Kai Hwang. *Parallel Processing for Supercomputers and Artificial Intelligence*. Mcgraw-Hill, March 1989.

[IA93] P. Idestam-Almquist. *Generalisation of Clauses*. PhD thesis, Stockholm University, 1993.

[JKP94] George H John, Ron Kohavi, and Karl Pfleger. Irrelevant Features and the Subset Selection Problem. pages 121—129, 1994.

[KMLS92] R D King, S Muggleton, R A Lewis, and M J Sternberg. Drug Design by Machine Learning: the Use of Inductive Logic Programming to Model the Structure-Activity Relationships of Trimethoprim Analogues Binding to Dihydrofolate Reductase. *Proceedings of the National Academy of Sciences of the United States of America*, 89(23):11322–11326, December 1992.

[Kow79] Robert Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[Lan94] Pat Langley. Selection of Relevant Features in Machine Learning. *In Proceedings of the AAAI Fall Symposium on Relevance*, pages 140—144, 1994.

[LCD92a] N. Lavrač, B. Cestnik, and S. Džeroski. Search Heuristics in Empirical Inductive Logic Programming. In C. Rouveirol, editor, *Proceedings of the ECAI-92 Workshop on Logical Approaches to Machine Learning*, 1992.

[LCD92b] N. Lavrač, B. Cestnik, and S. Džeroski. Use of Heuristics in Empirical Inductive Logic Programming. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, Report ICOT TM-1182, 1992.

[LD92] Nada Lavrac and Saso Dzeroski. Background Knowledge and Declarative Bias in Inductive Concept Learning. In *Proceedings of the International Workshop on Analogical and Inductive Inference*, pages 51–71. Springer-Verlag, 1992.

[LD94] Nada Lavrac and Saso Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Prentice Hall, 1994.

[LD10] Ailsa H. Land and Alison G. Doig. An Automatic Method for Solving Discrete Programming Problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. 2010.

[LDG91] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning Nonrecursive Definitions of Relations with LINUS. In *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 265–281. Springer-Verlag, 1991.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[LM92] S. Lapointe and S. Matwin. Sub-unification: A Tool for Efficient Induction of Recursive Programs. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Machine Learning Conference*, pages 273–281. Morgan Kaufmann, 1992.

[Low76] Bruce T. Lowerre. *The Harpy Speech Recognition System*. PhD thesis, Carnegie Mellon University, 1976.

[LWZ$^+$96] Nada Lavrač, Irene Weber, Darko Zupanič, Dimitar Kazakov, Olga Štěpánková, and Sašo Džeroski. ILPNET Repositories on WWW: Inductive Logic Programming Systems, Datasets and Bibliography. *AI Communications*, 9(4):157–206, 1996.

[MB88] S. Muggleton and W. Buntine. Machine Invention of First Order Predicates by Inverting Resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–351. Morgan Kaufmann, San Mateo, CA, 1988.

[McC92] William McCune. Experiments with Discrimination-tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992.

[MdR94] Stephen Muggleton and Luc de Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19:629–679, 1994.

[MF90] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[Mic83] R. Michalski. A Theory and Methodology of Inductive Learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *An Artificial Intelligence Approach*, volume 1, pages 83–134. Tioga, Palo Alto, CA, 1983.

[Mic88] D. Michie. Machine Learning in the Next Five Years. *Proceedings of the Third European Working Session on Learning*, pages 107–122, 1988.

[Min89] John Mingers. An Empirical Comparison of Selection Measures for Decision-tree Induction. *Machine Learning*, 3(4):319–342, March 1989.

[MISI98] Tohgoroh Matsui, Nobuhiro Inuzuka, Hirohisa SEKI, and Hidenori Itoh. Comparison of Three Parallel Implementations of an Induction Algorithm. *In 8th International Parallel Computing Workshop*, pages 181—188, 1998.

[Mit80] Tom M Mitchell. The Need for Biases in Learning Generalizations. pages 184—191, 1980.

[Mit82] T. Mitchell. Generalization as Search. *Artificial Intelligence*, 18(2):203–226, 1982.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, October 1997.

[MKKC86] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1:47–80, 1986.

[MKS92] S. H. Muggleton, R. King, and M. Sternberg. Protein Secondary Structure Prediction Using Logic. In *Proceedings of the Second International Workshop on Inductive Logic Programming*, Tokyo, Japan, 1992. ICOT TM-1182.

[MMHL86] R. Michalski, I. Mozetič, J. Hong, and N. Lavrač. The Multi-purpose Incremental Learning System AQ15 and its testing application on three medical domains. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 1041–1045. Morgan Kaufmann, San Mateo, CA, 1986.

[Moz87] Igor Mozetic. Learning of Qualitative Models. In *Progress in Machine Learning*, pages 201–217, 1987.

[MP94] S. Muggleton and C. D. Page. Self-saturation of Definite Clauses. In S. Wrobel, editor, *Proceedings of the Fourth International Inductive Logic Programming Workshop*, volume 237 of *GMD-Studien*, pages 161–174. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[MP98] S. Muggleton and D. Page. A Learnability Model for Universal Representations and its Application to Top-Down Induction of Decision Trees. *Machine Intelligence 15*, 1998.

[MSB92] Stephen Muggleton, Ashwin Srinivasan, and Michael Bain. Compression, Significance and Accuracy. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 338–347, 1992.

[MT91] R. Michalski and G. Tecuci, editors. *Proceedings of the First International Workshop on Multistrategy Learning*. George Mason University, Fairfax, 1991.

[Mug90] Stephen Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison Wesley, May 1990.

[Mug91] Stephen Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[Mug92] S. Muggleton. *Inductive Logic Programming*. Academic Press Inc, June 1992.

[Mug94] S. Muggleton. Predicate Invention and Utilization. *Journal of Experimental and Theoretical Artificial Intelligence*, 6(1):127–130, 1994.

[Mug95] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special Issue on Inductive Logic Programming*, 13(3–4):245–286, 1995.

[Mug97] Stephen Muggleton. Learning from positive data. In *Inductive Logic Programming*, pages 358–376. 1997.

[MWKE93] Katharina Morik, Stefan Wrobel, Jorg-Uwe Kietz, and Werner Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications*. Academic Press, 1 edition, September 1993.

[NB86] T. Niblett and I. Bratko. Learning Decision Rules in Noisy Domains. In M. A. Bramer, editor, *Research and Development in Expert Systems III*, pages 24–25, Cambridge, 1986. Cambridge University Press.

[NdW97] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., 1997.

[Nib88] T. Niblett. A Study of Generalisation in Logic Programs. In D. Sleeman, editor, *Proceedings of the Third European Working Session on Learning*, pages 131–138. Pitman, 1988.

[NRB$^+$96] Claire Nedellec, Céline Rouveirol, Francesco Bergadano, Birgit Tausend, and Fakultat Informatik. Declarative Bias in ILP. *Advances in Inductive Logic Programming*, 1996.

[Ohl90] Hans Jürgen Ohlbach. Abstraction Tree Indexing for Terms. 1990.

[ONM00] Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment. In *Inductive Logic Programming*, pages 165–173. 2000.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, April 1984.

[PFZ99] Nada Lavrac Peter, Peter Flach, and Blaz Zupan. Rule Evaluation Measures: A Unifying View. *Proceedings of the 9th International Workshop on Inductive Logic Programming (ILP-99)*, pages 174—185, 1999.

[Pla08] Plato. *Defence of Socrates, Euthyphro, Crito*. Oxford Paperbacks, May 2008.

[Plo70] G. D. Plotkin. A Note on Inductive Generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Plo71] G. D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, 1971.

[PMM71] Gordon Plotkin, Bernard Meltzer, and Donald Michie. A Further Note on Inductive Generalization. *Machine Intelligence*, 6:124, 101, 1971.

[PN91] Doug Palmer and Lee Naish. NUA-Prolog: An Extension to the WAM for Parallel Andorra. In Koishi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 429–442, 1991.

[PSCF03] David Page, Ashwin Srinivasan, James Cussens, and Alan Frisch. ILP: A Short Look Back and a Longer Look Forward. *Journal of Machine Learning Research*, 4:415—430, 2003.

[QC93] J. R Quinlan and R. M Cameron-Jones. FOIL: A Midterm Report. In *Proceedings of the European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 3–20, 1993.

[Qui] J. R. Quinlan. Discovering Rules from Large Collections of Examples: A Case Study. *Expert Systems in the Micro-electronic Age*.

[Qui86] J. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[Qui90] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[Rae92] Luc De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London, September 1992.

[RB93] L. De Raedt and M. Bruynooghe. A Theory of Clausal Discovery. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 25–40. Stefan Institute, 1993.

[RD96] Luc De Raedt and Luc Dehaspe. Clausal Discovery. *Machine Learning*, 26:1058—1063, 1996.

[RK03] Luc De Raedt and Kristian Kersting. Probabilistic Logic Learning. *ACM SIGKDD Explorations Newsletter*, 5(1):31–48, 2003.

[RK04] Luc De Raedt and Kristian Kersting. Probabilistic Inductive Logic Programming. *Proceedings of the 15th International Conference on Algorithmic Learning Theory*, 3244, 2004.

[RL93] L. De Raedt and N. Lavrač. The Many Faces of Inductive Logic Programming. In J. Komorowski and Z. W. Raś, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, volume 689 of *Lecture Notes in Artificial Intelligence*, pages 435–449. Springer-Verlag, 1993. (Invited paper).

[RN02] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2 edition, December 2002.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[Rou92] C. Rouveirol. Extensions of Inversion of Resolution Applied to Theory Completion. In S. Muggleton, editor, *Inductive Logic Programming*, pages 63–92. Academic Press, London, 1992.

[rpr] The R Project for Statistical Computing. http://www.r-project.org/.

[RRS$^+$99] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S Warren. Efficient Access Mechanisms for Tabled Logic Programs. 1999.

[RSFC00] Ricardo Rocha, Fernando Silva, Ricardo Rocha Fern, and Vítor Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. 2000.

[Sam84] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Sha91] Ehud Y. Shapiro. Inductive Inference of Theories from Facts. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 199–254, 1991.

[Sha05] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, July 2005.

[Sil09] Abraham Silberschatz. *Operating System Concepts*. John Wiley & Sons, 8th edition, February 2009.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[Smi88] R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. In *Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers Inc., 1988.

[SR97] Michèle Sebag and Céline Rouveirol. Tractable induction and classification in first order logic via stochastic matching. 1997.

[Sri] Ashwin Srinivasan. The Aleph Manual. http://www.comlab.ox.ac.uk/activities/machinelearning/Alep

[Sri99] Ashwin Srinivasan. A Study of Two Sampling Methods for Analysing Large Datasets with ILP. 1999.

[Ste94] L. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. MIT Press, 2nd revised edition, April 1994.

[SW] Terrance Swift and David S Warren. Analysis of SLG-WAM Evaluation of Definite Programs. *Proceedings of the International Logic Programming Symposium 1994*, pages 219—235.

[Tur39] A. M. Turing. Systems of Logic Based on Ordinals. *Proceedings of the London Mathematical Society*, pages 161–228, 1939.

[Tur47] A. M. Turing. The Automatic Computing Engine. *Lecture to the London Mathematical Society*, 1947.

[Ull88] Jeffrey D. Ullman. *Principles of Database & Knowledge-Base Systems Vol. 1: Classical Database Systems*. Computer Science Press, 1 edition, December 1988.

[UM82] P. Utgoff and T. M. Mitchell. Acquisition of Appropriate Bias for Inductive Concept Learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 414–417. Morgan Kaufmann, Los Altos, 1982.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. *Technical Note 309*, 1983.

[War95] David S Warren. Programming in Tabled Prolog. 1995.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2 edition, July 2005.

[Wie03] Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, december 2003. Springer Verlag. LNCS 2916.

[Wir89] R. Wirth. Completing Logic Programs by Inverse Resolution. In K. Morik, editor, *Proceedings of the 4th European Working Session on Learning*. Pitman, London, 1989.

[ZSP02] Filip Zelezny, Ashwin Srinivasan, and David Page. Lattice-Search Runtime Distributions May Be Heavy-Tailed. *In Proceedings of the 12th International Conference on Inductive Logic Programming*, pages 333—345, 2002.