Imperial College of Science, Technology and Medicine
Department of Computing

Efficient Learning and Evaluation of Complex Concepts in Inductive Logic Programming

José Carlos Almeida Santos

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College, December 2010

Abstract

Inductive Logic Programming (ILP) is a subfield of Machine Learning with foundations in logic programming. In ILP, logic programming, a subset of first-order logic, is used as a uniform representation language for the problem specification and induced theories. ILP has been successfully applied to many real-world problems, especially in the biological domain (e.g. drug design, protein structure prediction), where relational information is of particular importance.

The expressiveness of logic programs grants flexibility in specifying the learning task and understandability to the induced theories. However, this flexibility comes at a high computational cost, constraining the applicability of ILP systems. Constructing and evaluating complex concepts remain two of the main issues that prevent ILP systems from tackling many learning problems. These learning problems are interesting both from a research perspective, as they raise the standards for ILP systems, and from an application perspective, where these target concepts naturally occur in many real-world applications. Such complex concepts cannot be constructed or evaluated by parallelizing existing top-down ILP systems or improving the underlying Prolog engine. Novel search strategies and cover algorithms are needed.

The main focus of this thesis is on how to efficiently construct and evaluate complex hypotheses in an ILP setting. In order to construct such hypotheses we investigate two approaches. The first, the Top Directed Hypothesis Derivation framework, implemented in the ILP system TopLog, involves the use of a top theory to constrain the hypothesis space. In the second approach we revisit the bottom-up search strategy of Golem, lifting its restriction on determinate clauses which had rendered Golem inapplicable to many key areas. These developments led to the bottom-up ILP system ProGolem. A challenge that arises with a bottom-up approach is the coverage computation of long, non-determinate, clauses. Prolog's SLD-resolution is no longer adequate. We developed a new, Prolog-based, theta-subsumption engine which is significantly more efficient than SLD-resolution in computing the coverage of such complex clauses.

We provide evidence that ProGolem achieves the goal of learning complex concepts by presenting a protein-hexose binding prediction application. The theory ProGolem induced has a statistically significant better predictive accuracy than that of other learners. More importantly, the biological insights ProGolem's theory provided were judged by domain experts to be relevant and, in some cases, novel.

Acknowledgements

I would like to thank all who made this thesis possible, starting with my supervisor Stephen Muggleton. Stephen's world-class expertise in Inductive Logic Programming and his broad knowledge of science had a profound impact on this PhD and on my education. Furthermore, his generosity with the meeting times and prompt feedback on early drafts of this thesis were key factors for finishing this PhD on time. To my second supervisor, Michael Sternberg, I would like to thank the trust deposited by selecting me for this PhD programme. Michael's vast and deep knowledge of proteins and ILP applications was also very important to Chapter 7.

I want to acknowledge the Wellcome Trust who, under the grant 0807/12/Z/06/Z, generously funded my four-year PhD programme.

The implementation work of this dissertation relied heavily on the YAP Prolog compiler. I want to thank YAP's author, Vítor Santos Costa, for his prompt replies to my innumerous e-mails, quickly fixing bugs and introducing some features that were essential for this research.

I was fortunate enough to be in a research group, the Computational Bioinformatics group at Imperial College, which hosts both friendly and bright research assistants and PhD students, nurturing the ideal work environment for research. I want to acknowledge the following colleagues from our research group with whom I enjoyed sharing ideas over these years: Niels Pahlavi, Robert Henderson, Jianzhong Chen, Dianhuan Lin, Alireza Tamaddoni-Nezhad, Hiroaki Watanabe, Aline Paes, Huma Lodhi, Robin Baumgarten, Ramin Ramezani, Pedro Torres and John Charnley. Of these, I want to highlight especially Alireza, with whom I had the pleasure to closely collaborate on several papers which led to Chapters 3 and 4.

I want to acknowledge Ondrej Kuzelka and Filip Zelezny, with whom I had lively e-mail discussions over the performance of subsumption engines. This healthy competition and exchange of ideas provided the motivation to further improve Subsumer (Chapter 5). I want to thank Houssam Nassif for his collaboration in the protein-hexose application which led to Chapter 7. I would also like to thank Suhail Islam for his help in generating the nice protein-hexose binding site picture, Figure 7.8, of Chapter 7.

Finally, I warmly thank my wife Gilda Ferreira for her love and support over the years. Her presence makes my life much more enjoyable and worthwhile.

Statement of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own except where otherwise stated.

To my beloved wife Gilda

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Inductive Logic Programming (ILP) is a subfield of Machine Learning with foundations in logic programming. In ILP, logic programming, a subset of first-order logic, is used as a uniform representation language for the problem specification and induced theories. ILP has been successfully applied to many real-world problems, especially in the biological domain (e.g. drug design [KMLS92], protein structure prediction [MKS92]), where relational information is of particular importance.

The expressiveness of logic programs grants flexibility in defining the learning task and understandability to the induced theories. However, this flexibility comes at a high computational cost and ILP systems are known for their difficulty in scaling-up [PS03]. There are several aspects to this scaling-up problem, an important one being the presence of highly non-determinate background knowledge. The non-determinacy of the background knowledge is particularly an issue with structural biology problems, to which we will pay particular attention in this thesis.

The sheer amount of data that needs to be processed is another aspect to the scaling-up problem. In the last decade much work has been done to make ILP systems more efficient by improving the performance of the underlying Prolog compiler [dSC06]. In particular, several ILP systems, including ours, are implemented in YAP Prolog [Cos09], which has been specifically improved to support ILP development.

Another issue in scaling-up ILP is the large size of the hypothesis space. The initial approach we took to reduce the hypothesis space was to study a more stringent search bias than the one provided by the usual mode declarations. We revisited previous work in declarative search bias [Coh94] and developed the Top Directed Hypothesis Derivation (TDHD) framework. TDHD further restricts the hypothesis space by requiring it to be entailed by a Top theory. This framework was implemented in the TopLog ILP system [MSTN08].

However, constructing and evaluating complex concepts remain two of the main issues that prevent ILP systems from tackling many learning problems. Problems whose target concept is complex are interesting both from a research and application perspective. From a research perspective these raise the standards for ILP systems; from an application perspective these target concepts naturally occur in many real-world applications. Despite the merits in parallelizing the hypothesis search [Fon06], such complex concepts cannot be constructed or evaluated by parallelizing existing top-down ILP systems or improving the underlying Prolog engine. Novel search strategies and cover algorithms are needed.

The main focus of this thesis is on how to efficiently construct and evaluate complex hypotheses in an ILP setting. In order to be able to construct complex hypotheses we revisit the bottom-up search strategy of Golem [MF92], lifting its restriction on determinate clauses which had rendered Golem inapplicable to many key areas.

We allow for non-determinate clauses by exploring a variant of Plotkin's relative (to a Progol's bottom clause) least general generalization ($rlgg$) [Plo71] where efficient refinement operators can be implemented. In contrast to Plotkin's $rlgg$, where clause length grows exponentially in the number of examples, in our framework the clause length decays with the number of examples and is provably bounded by the length of the initial bottom clause [MSTN09].

These ideas, combining the bottom-up search of Golem with Progol's [Mug95b] bottom clause bound, were implemented in a new ILP system, ProGolem [MSTN09], capable of learning long, non-determinate, clauses.

A challenge that arises when we have long, non-determinate, hypotheses is computing their

coverage efficiently. Prolog's SLD-resolution is no longer adequate. In this respect, a significant contribution of this thesis is a novel state-of-the-art theta-subsumption algorithm, Subsumer [SM10a], which is significantly more efficient than SLD-resolution. In practice, in challenging datasets such as the Phase Transition [GS00], this translates to several orders of magnitude speed-up on performing coverage testing.

Efficient theta-subsumption engines are not only important to ILP; these engines have a far-reaching impact on the wider computational logic community, e.g. AI planning [Skv06].

On the datasets to which Aleph [Sri07], a state-of-the-art ILP system, is applicable, ProGolem often has comparable or better performance (i.e. lower running times and/or higher predictive test accuracy). Furthermore, ProGolem can be applied to a wider range of learning problems, sometimes revealing more insightful theories. A compelling example for the latter claim is a recent application of ProGolem to a protein-binding dataset, where the theory found by ProGolem led to novel scientific knowledge (see Section 7.3.3).

1.1 Objectives and Main Contributions

The broad objective of our work is to advance the state of ILP systems. The main contributions of this dissertation are:

1. Top-directed declarative search bias: see [MSTN08] and Chapter 3.

2. A novel, efficient bottom-up ILP learner, ProGolem: see [MSTN09] and Chapter 4.

3. A new, Prolog-based, efficient theta-subsumption engine, Subsumer: see [SM10a] and Chapter 5.

4. Demonstration that ProGolem can be applied to a wider range of learning problems than other state-of-the-art ILP systems: see [MSTN09] and Chapters 4 and 7.

5. GILPS: General Inductive Logic Programming System that implements all the above contributions. See Chapter 6 for a description of the system, its main features and a tutorial. The software is publicly available at http://ilp.doc.ic.ac.uk/GILPS.

In addition to its applicability to ILP, Subsumer has potential to be useful to the wider computational logic community, as efficient subsumption engines are important to several fields, e.g. AI planning [Skv06]. Outside of Computer Science, a noteworthy contribution of this PhD is to molecular biology where the rules ProGolem induced provided revealing insights to an important protein-hexose binding prediction problem (Chapter 7).

Handling uncertainty in a relational framework has been gaining increased attention in Machine Learning. This led to the development of Statistical Relational Learning (SRL) [GT07] where ideas from probability theory and statistics are incorporated with logic models. Combining probabilities with ILP has also been a topic of research in recent years [RFKM08].

However, this dissertation is focused solely on the logic side of ILP. The main reason for not having pursued a probabilistic route is because we believe a bigger impact could be made by focusing on the fundamental difficulties of structure learning, such as being able to efficiently learn and evaluate complex concepts. Since SRL builds on structure learning techniques, such as those developed within ILP, we are also potentially contributing towards better SRL systems, by improving the ILP framework so that it is able to deal with a wider range of learning problems.

1.2 Publications

The following publications arose from work conducted during the course of this PhD:

1. "TopLog: ILP Using a Logic Program Declarative Bias" [MSTN08] presented at the 24th International Conference in Logic Programming (ICLP08). Chapter 3 presents a significantly extended version of this work. The initial framework we developed then is

currently being extended by a PhD student in our group, Dianhuan Lin, to allow for multi-clause learning.

2. "An ILP System for Learning Head Output Connected Predicates" [STNM09] presented at the 14th Portuguese Conference on Artificial Intelligence (EPIA09). This paper presents an efficient special-purpose ILP system to learn target theories which have output variables in the head of a clause. It is an interesting work but does not fit into the main argument of the thesis.

3. "ProGolem: a system based on relative minimal generalisation" [MSTN09] presented at the 19th International Conference on Inductive Logic Programming (ILP09). The theory underlying asymmetric relative minimal generalizations (*armg*s) is the authorship of Alireza Tamadonni-Nezhad and Stephen Muggleton. Our work was the implementation of the *armg* concept in a bottom-up ILP system, ProGolem. This paper is the basis for Chapter 4.

4. "Subsumer: A Prolog theta-subsumption engine" [SM10a] presented at the 26th International Conference in Logic Programming (ICLP10). This paper presents an efficient subsumption engine that is useful to the wider computational logic community and particularly relevant to ProGolem. An extended version is presented in Chapter 5.

5. "When does it pay off to use sophisticated entailment engines in ILP?" [SM10b] presented at the 20th International Conference in Logic Programming (ILP10). This short paper presents and benchmarks the four coverage engines developed within ProGolem. Its main results are presented in Section 6.2.2.

In work carried out prior to the choice of the thesis topic, the author of this thesis also co-authored the paper "Learning probabilistic logic models from probabilistic examples" [CMS08], where a probabilistic approach to examples in ILP was explored. However, for the reasons explained in Section 1.1, the development of a probabilistic approach to ILP was not the subject of this thesis.

1.3 Thesis outline

In Chapter 2 we present a background overview on Machine Learning and ILP.

Chapter 3 presents the TopLog ILP system. TopLog is, like Progol or Aleph, a top-down ILP system; however, unlike those, it allows for a Top-directed declarative search bias.

Chapter 4 presents ProGolem, a bottom-up ILP system capable of learning long and complex concepts. One of the main challenges of a bottom-up ILP system is evaluating the coverage of the long clauses generated in the early stages of the hypotheses search.

This challenge was the initial motivation to develop a series of entailment engines that culminated in the development of an efficient theta-subsumption engine, Subsumer. Subsumer is presented in Chapter 5.

In Chapter 6 we describe GILPS, the modular and highly configurable ILP system that implements all the algorithms and systems described in this dissertation.

Chapter 7 is a real-world application of ProGolem to the problem of predicting protein-hexose binding. Crucial to the success of this application are the bottom-up search strategy of ProGolem and the usage of efficient coverage engines.

We conclude with Chapter 8 where a summary of the achievements and ideas for future directions are presented.

Chapter 2

Machine Learning and ILP Overview

In this chapter we introduce some background concepts on the broader areas of Machine Learning, Logic Programming and Inductive Logic Programming. ILP is at the intersection of Machine Learning and Logic Programming. The aim of this overview is twofold: to be self-contained by providing the unfamiliar reader with enough background to understand this thesis and to put ILP and our contributions to it into perspective. For a detailed overview we recommend [Mit97] for Machine Learning, [Llo87] for Logic Programming and [NCdW97] for Inductive Logic Programming.

Machine Learning is a broad sub-field of Artificial Intelligence; its aim is to devise algorithms that allow automated learning. Machine Learning is divided into two main areas: unsupervised and supervised learning.

In unsupervised learning, the instances are unlabelled and the learner seeks to determine how the data is organized. A common task in unsupervised learning is clustering, where the goal is to organize the given data in a set of distinct clusters whose instances are more similar within a cluster than with instances from other clusters.

In supervised learning, the training data consists of instances labelled with the desired target output. The task is to learn a function that generalizes from the supplied examples to unseen ones. This generalization is needed in order to classify unseen instances. For instance, a super-

vised learning problem may be to learn the concept of "bird" through observing a small labelled sample of animals, each with a given set of attributes (e.g. has_feathers, flies, drinks_milk).

When learning the target concept, the learning algorithm is presented with a set of training examples, each consisting of an instance e from the set of possible examples E, along with its target concept value $c(e)$. The problem the learner solves is to find a hypothesis, h, such that $h(e) = c(e)$. Thus, learning involves a search through a space of possible hypotheses to find the hypothesis that best fits the available training examples and other prior knowledge or constraints.

Notice that because the only information available about the target concept c is its value over the training examples, all learning algorithms have to assume that any hypothesis that approximates the target concept well over a sufficiently large set of training examples will also approximate the target function well over unobserved examples. This assumption, that training and test examples are drawn from the same distribution, is central to machine learning.

Our work with ILP in this thesis is always in the supervised learning setting. Hereafter, we will refer to supervised learning simply as learning. Before we introduce ILP we will first start with a brief introduction to Logic Programming.

2.1 Logic Programming

Logic Programming is a programming paradigm based on a subset of first-order logic. First-order logic is a rich formal deductive system with far more expressive power than propositional logic (which is used, for instance, in classical decision trees). Its added expressiveness allows us to distinguish between "things" and assertions about "things", denoting the same "thing" (term) and concept (predicate) everywhere by the same symbol.

The subset of first-order logic used in Logic Programming consists of first-order Horn clauses. Horn clauses are logic clauses with at most one positive literal. Horn clauses with exactly one positive literal are called definite clauses.

It is often convenient to see a definite clause as an implication instead of a disjuction of literals. For instance, the clause $\neg p(X) \vee \neg q(X) \vee \neg t(X) \vee h(X)$ is equivalent to the implication $(p(x) \wedge q(X) \wedge t(X)) \rightarrow h(x)$. In the form of an implication definite clauses have a straightforward procedural interpretation, e.g. to show $h(X)$, show $p(X)$, $q(X)$ and $t(X)$.

Definition 2.1. Definite clause *A* definite clause *is a clause of the form* $H \leftarrow B_1, \ldots, B_n$ *which contains precisely one literal* (H) *in its consequent and 0 or more literals* ($B_1, \ldots, B_n$) *as the antecedent.* H *is called the head and* $B_1, \ldots, B_n$ *is called the body of the clause.*

The head of the clause holds (i.e. is true) if there is an assignment of values that makes all the body literals true simultaneously. For instance, consider the clause $h(X) \leftarrow b(X, Y), c(Y)$ and the following facts: $\{b(1, 2).b(2, 3).c(2).\}$. The only assignment that satisfies the clause is $X = 1$ and $Y = 2$, allowing us to deduce $h(1)$.

The reason to restrict Prolog to Horn clauses is that, as shown by [Kow74], it allows for an efficient proof procedure, namely Selective Linear Definite clause resolution (SLD-resolution) [KK71].

SLD-resolution is sound and refutation complete for Horn clauses [Llo87]. Within SLD-resolution, a refutation of a goal G within a program P consists in deriving the empty clause, $\square$, as the last goal in the derivation. A refutation fails when a subgoal fails to unify with the head of any clause in P. By refutation complete it is meant that, if a refutation exists, it will be found [1] in a finite number of steps. If a refutation of a goal G is found, then we can conclude $\neg G$.

The sequence of Horn clauses derived by applying SLD-resolution is called an SLD-derivation. We formally define SLD-derivation below and provide a simple example.

Definition 2.2. SLD-derivation *Let* $C_1, C_2, \ldots, C_n$ *be definite clauses and* G_0 *be a Horn clause. An SLD-derivation, denoted by* $R = \langle G_0, C_1, \ldots, C_n \rangle$*, is a sequence of Horn clauses* $G_0, G_1, \ldots, G_n$ *such that for each* $1 \leq i \leq n$*,* G_i *is a binary resolvent of* G_{i-1} *and* C_i*, using the head of* C_i *and a literal selected from the body of* G_{i-1} *as the literals resolved upon. We say that* R *derives the Horn clause* G *in the case that* G *is the final resolvent of* R*.*

[1] In Prolog this does not hold because of Prolog's depth-first search leftmost computation rule, which can lead to infinite recursion. Also, for efficiency reasons, Prolog does not implement the occurs check.

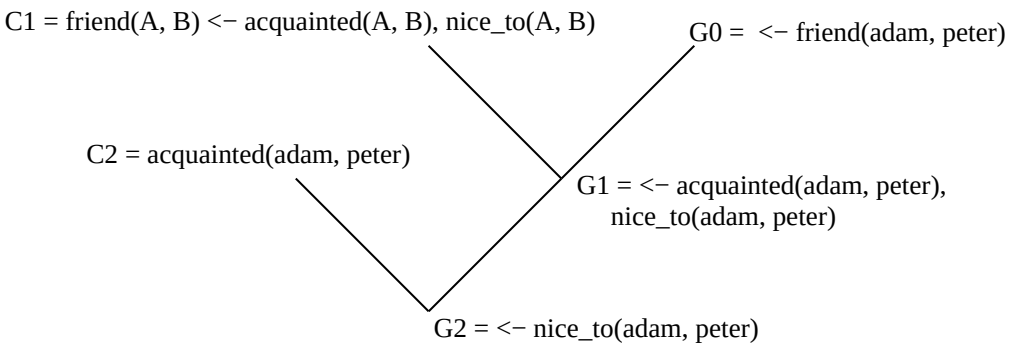


Figure 2.1: Example of SLD-derivation

Definition 2.3. SLD-refutation *An SLD-derivation which derives the empty clause, $\square$, is called an SLD-refutation.*

Example 2.4. SLD-derivation *Figure 2.1 shows the SLD-derivation $R = \langle G_0, C_1, C_2 \rangle$. Clauses G_0 and C_1 are first resolved on their leftmost respective literals to give the resolvent G_1. Similarly in the second resolution C_2 is resolved with G_1 to give the final resolvent G_2. Thus R derives G_2.*

In contrast to imperative programming languages, in Logic Programming programs are expressed as a set of clauses and facts (i.e. bodiless clauses). SLD-resolution is then used to mechanically prove queries and, as a by-product, bind logic variables to values that satisfy the query.

Prolog, initially created in France in 1972 by Alain Colmerauer, is the general-purpose programming language implementing the Logic Programming paradigm. Prolog is also a declarative language in the sense that the programmer only needs to express what is known about the problem domain (i.e. facts and relationships between terms) and the built-in resolution mechanism takes care of the control part. Prolog is mainly used in artificial intelligence applications. Its uses are often in the areas of knowledge representation, constraint logic programming, natural language processing and inductive logic programming.

2.1.1 A note on Prolog compilers

There are a number of open source and commercial Prolog compilers. Currently, the most well-known commercial Prolog is Sicstus, the descendant of Quintus Prolog. As for open source implementations, SWI-Prolog is the best known with the largest number of users. Though robust, SWI is not a particularly efficient Prolog implementation, being several times slower than Sicstus in many benchmarks. A less-known but mature and well-documented Prolog compiler is YAP [Cos09]. YAP is one of the fastest open source Prologs, even faster than Sicstus in many key applications.

The initial versions of the general ILP system we developed for this project, GILPS (see Chapter 6), supported Sicstus, SWI and YAP but it was becoming a growing burden to keep compatibility between the three Prologs as all deviate slightly from the ISO and have a slightly different set of built-in libraries.

We opted to support exclusively YAP for several reasons. One of the YAP's design goals has been to facilitate the development of ILP systems, having built-in support for e.g. non-backtrackable data-structures, resolution counters and depth-bounded calls, that greatly simplify development. It also has the best performance due to features such as demand-driven indexing of clauses [CSL07]. From our experience, a rough figure for the relative performance of these Prolog compilers, using ILP systems as benchmarks, is that SWI is approximately 10 times slower than Sicstus and Sicstus is, in turn, approximately 10 times slower than YAP.

2.2 Inductive Logic Programming

2.2.1 The normal ILP setting

The purpose of Inductive Logic Programming (ILP) is, in its simplest form, to induce (i.e. discover) the definition of a (target) predicate by observing positive and negative examples of it. That is, instances where the predicate holds and where the predicate does not hold are

provided and labelled as such.

Together with positive and negative examples of the target predicate, other background information (i.e. clauses and facts) may also be provided containing further information relevant to learning the target predicate. This background information is also a logic program and is called the *background knowledge* in ILP. See Section 2.2.3 for an example.

The goal of an ILP system is to learn a definition of the target predicate such that, together with the background knowledge, positive examples are entailed but negative examples are not. More formally, according to [NCdW97], the normal ILP setting is defined as:

Given: a finite set of clauses B (background knowledge) and two disjoint sets of clauses E^+ and E^- (positive and negative examples)

Find: a theory H (i.e. a set of clauses) such that $H \cup B$ is correct with respect to E^+ and E^-.

A theory H is complete with respect to E^+, if $H \models E^+$. H is consistent with respect to E^-, if $H \not\models E^-$. A theory which is both complete with respect to E^+ and consistent with respect to E^- is called correct. Inducing a correct theory is the ultimate purpose of an ILP system, but in real-world problems this is often not possible due to noise in the examples or errors in the background knowledge.

Notice that the trivial (and lengthy) theory $H = E^+$ would always be correct but is of zero predictive power, since no concept has been learned and thus any unseen example will always be classified as negative. The learning goal is to find a theory that generalizes from the provided examples to unseen ones, thus having good predictive power. A good heuristic to achieve this goal is Occam's Razor. Its application to learning theory is the minimum description length principle where, example coverage being equal, one prefers shorter to longer theories.

In the spirit of the minimum description length principle, ILP systems typically evaluate the merit of a theory by a compression measure. The compression of a theory is given by the weight of all the positive examples it covers minus the weight of the negative examples covered

minus the size (in number of literals) of the theory itself. A theory is thus only compressive if it encodes the examples using fewer literals than the ones originally needed.

There are two main approaches to finding a correct theory H: top-down and bottom-up. A top-down approach starts with an overly general theory and specializes it to cover fewer examples. With a bottom-up approach the starting theory is overly specific and is generalized to cover more examples. In both cases an important concept is that of refinement operator. A refinement operator performs a change (small or large) in a clause in order to generalize or specialize it.

An example of a specialization refinement operator is to append a literal to the body of a clause. An example of a generalization refinement operator is to delete a literal from the body. For the refined clause to be valid one needs to ensure all its literals are head-connected (see Definition 4.7 in Chapter 4).

For the specialization refinement operator, the head-connectedness requirement implies that the appended new literal must have its input arguments bound to already existing variables in the clause. For the generalization refinement operator it is required that, when removing a literal l, those other literals that have, as input variables, output variables generated exclusively by l, must be removed as well. See next section for an explanation of input/output arguments of a literal in the context of ILP.

2.2.2 Defining the hypothesis search space: mode declarations

The purpose of mode declarations in an ILP system is to bias and delimit the hypothesis search space. Mode declarations characterize the format of a valid hypothesis. There are two types of mode declarations: head and body. Mode head declarations, *modeh*, state which is the target predicate the ILP system is supposed to induce (i.e. the head of a valid hypothesis) and mode body declarations, *modeb*, state which literals may appear in the body of such hypotheses. Normally mode body declarations refer to predicates defined in the background knowledge but, in the case of recursive theories, they can also refer to the target predicate being induced.

In Prolog it is usually assumed a predicate can be called with any possible combination of its

arguments instantiated. However, this assumption is not always verified as predicates often require a specific subset of its arguments to be ground before the predicate is callable. Thus, mode declarations also inform on the types and input/output modes of the arguments of the predicates that will appear in a hypothesis.

In the mode declarations, each argument of a predicate has a type and an associated prefix, either '+', '-' or '#'. The prefix '+' means that the argument is an input argument (i.e. the argument must be instantiated before the predicate is called), '-' means it is an output argument (i.e. the argument does not need to be instantiated before the predicate is called) and '#' means it is a constant. Constant arguments are like output arguments except that, instead of returning a variable that may be used later as input to another predicate, a constant is yielded directly in the hypothesis body.

By typing the arguments of the predicates and imposing input/output restrictions we guarantee that the clauses generated as hypothesis are at least executable by the Prolog engine. Note that these restrictions also reduce the hypothesis space considerably.

A Prolog predicate may be determinate or non-determinate. A predicate is determinate if it succeeds at most once given a particular instantiation of its input arguments, i.e. is a function. A non-determinate predicate may succeed more than once for a given instantiation of its input arguments. For instance, in the background knowledge of the Michalski's train problem, predicate *has_carriage/2* is non-determinate (a train may have more than one carriage) and all the others are determinate, e.g. in *wheels/2*, given a carriage there is only one possible number of wheels for that carriage.

Another item of information given by the mode declarations is the recall of a predicate. The recall of a predicate is the maximum number of times the predicate is allowed to succeed (i.e. its maximum number of solutions) when constructing the most-specific clause (see Section 2.2.4) of an example.

When the predicate is determinate the recall is 1, when it is non-determinate the recall is an integer greater than 1. Occasionally, in order to reduce the size of the hypothesis space, it

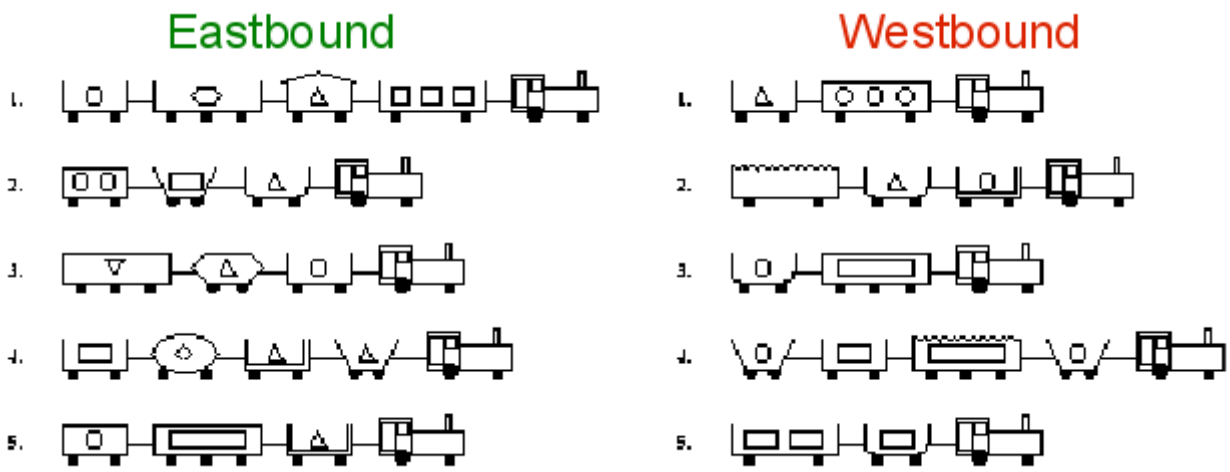


Figure 2.2: Michalski's trains problem

is useful to limit the non-determinacy of a predicate by reducing the number of solutions the predicate may return to a number lower than its actual number of possible solutions, e.g. if we set the recall of the *has_carriage/2* predicate to 3 only the first 3 carriages of a train would be returned. Note that limiting the recall of a predicate only impacts the most-specific clause construction, it does not restrict the number of solutions of the predicate elsewhere (i.e. the coverage of a clause is unchanged).

To illustrate the usage of mode declarations and the other concepts presented we will use the Michalski's trains problem [LM77] as a toy example. Figure 2.2 displays two sets of trains. The ones on the left travel east whereas the ones on the right travel west. The decision of whether a train is to travel east or west is made by a (unknown) rule that is a function of the train composition. The purpose of the Michalski problem is to find this unknown rule (i.e. the target concept).

The mode declarations allow us to formalize which literals may appear in a candidate hypothesis. Here we want those literals to be discriminative of the properties of a train and its carriages. A possible set of mode declarations is illustrated in Figure 2.3.

The *modeh* declaration says that the target predicate is *eastbound/1* and that it is called with an instantiated variable of type *train*. The *has_carriage/2* mode body declaration expects a term of type *train* as input and returns a term of type *carriage*. The '*' symbol at its beginning

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

Figure 2.3: Mode declarations for the Michalski's trains problem

means that the recall is unbounded[2], i.e. the predicate may succeed an arbitrary number of times. The *wheels/2* mode body declaration expects a term of type *carriage* and returns an integer constant, succeeding only once. The explanation is analogous for the remaining mode body declarations.

2.2.3 The examples and background knowledge

The labelled examples, as depicted in Figure 2.2, are translated to Prolog in Figure 2.4.

```
eastbound(east1).          :-eastbound(west1).
eastbound(east2).          :-eastbound(west2).
eastbound(east3).          :-eastbound(west3).
eastbound(east4).          :-eastbound(west4).
eastbound(east5).          :-eastbound(west5).
```

Figure 2.4: Examples for the Michalski's train problem

The :-/1 symbol means the succeeding fact is false. All negative examples are therefore preceded by :-/1.

The background knowledge to ILP is an arbitrary Prolog program. Note that background knowledge is not limited to facts about the examples. It can contain general concepts (e.g. arithmetic operations, distance metrics, ...) that may be useful to build the target theory.

In this Michalski's trains problem the background Prolog program is rather simple. It is simply a list of the facts that are true for each example. Figure 2.5 shows the background knowledge for example *east2* (second train on the left in Figure 2.2).

[2]In practice recall is bound to a large, usually user-defined, integer, e.g. in Progol it is 100.

```
has_carriage(east2,car_21).
has_carriage(east2,car_22).
has_carriage(east2,car_23).
infront(east2,car_21).
infront(car_21,car_22).
infront(car_22,car_23).
short(car_21).
short(car_22).
short(car_23).
shape(car_21,u_shaped).
shape(car_22,u_shaped).
shape(car_23,rectangle).
open(car_21).
open(car_22).
closed(car_23).
load(car_21,triangle,1).
load(car_22,rectangle,1).
load(car_23,circle,2).
wheels(car_21,2).
wheels(car_22,2).
wheels(car_23,2).
```

Figure 2.5: Background knowledge for example *east2*

2.2.4 Bounding the hypothesis search space: most-specific clause

An important concept in ILP is that of the most-specific clause, often called bottom clause or $\perp$. There is one most-specific clause per example, containing all facts known to be true about the example as derivable from the mode declarations. There are two versions of the most-specific clause, a ground and a variablized version. For instance, the ground most-specific clause for example *eastbound(east2)* is presented in Figure 2.6.

```
eastbound(east2):-
  has_carriage(east2,car_23), has_carriage(east2,car_22),
  has_carriage(east2,car_21), infront(east2,car_21), closed(car_23),
  open(car_22), open(car_21), short(car_23), short(car_22), short(car_21),
  load(car_23,circle,2), load(car_22,rectangle,1), load(car_21,triangle,1),
  wheels(car_23,2), wheels(car_22,2), wheels(car_21,2),
  infront(car_22,car_23), infront(car_21,car_22).
```

Figure 2.6: Ground most-specific clause for example: eastbound(east2)

After replacing each unique constant by a distinct variable, the ground most-specific clause of Figure 2.6 is transformed into the variablized most-specific clause of Figure 2.7. Note that the constants that come from a constant placeholder in the mode body declaration are not assigned a variable and remain the same constant. In this case, these are the integer constants appearing in literals *load/3* and *wheels/2*.

In this dissertation when we refer to $\perp$ we are referring to the variablized most-specific clause of an example. It is the variablized form of the most-specific clause that is used to bound

```
eastbound(A):-
   has_carriage(A,B), has_carriage(A,C),
   has_carriage(A,D), infront(A,D), closed(B),
   open(C), open(D), short(B), short(C), short(D),
   load(B,circle,2), load(C,rectangle,1), load(D,triangle,1),
   wheels(B,2), wheels(C,2), wheels(D,2),
   infront(C,B), infront(D,C).
```

Figure 2.7: Variablized most-specific clause for example: eastbound(east2)

the hypothesis space implicit in an example. In particular, in a top-down ILP system $\perp$ is at the bottom of the hypothesis space (thus the alternative name bottom clause), whereas in a bottom-up system $\perp$ is at the top of the search, being the first clause to be considered.

The concept of most-specific clause is therefore central to an ILP system. The algorithm to construct $\perp$, already in its variablized form, is presented in Figure A.2 of Appendix A.

One important, user-defined, parameter when constructing the most-specific clause is i, the number of layers of variables to consider. When $i = 1$ only the literals having, as input variables, input variables of the head (layer 0) are added to the most-specific clause. At layer i only literals having input variables appearing in layer $i - 1$ (as output or input variables) can be constructed. The output variables of the head, if any, can never be used as input variables for a literal unless they already appeared as the output variable of some other previous layer literal.

It is important to note that with a low value for i not all facts from the background knowledge will appear in a most-specific clause. To illustrate the impact of i, Figure 2.8 shows the ground most-specific clause of the same example, *eastbound(east2)*, when it is constructed with $i = 1$.

```
eastbound(east2):-
   has_carriage(east2,car_23), has_carriage(east2,car_22),
   has_carriage(east2,car_21), infront(east2,car_21).
```

Figure 2.8: Ground most-specific clause for example *eastbound(east2)* with i=1

The reason that all facts relating to example *eastbound(east2)* are present in the most-specific clause of Figure 2.6 is because a sufficiently high i value was used (in this case $i = 2$ suffices).

In Progol, Aleph and GILPS the parameter i is set to 3 by default. Notice that the length of the most-specific clause can potentially grow exponentially with i.

The choice of the value for i thus directly determines the size of the hypothesis space. With a low value for i the target concept may not be present in the hypothesis space as the required literals may not occur in the most-specific clause.

For instance, the target concept for the instance of the Michalski's trains problem presented in Figure 2.2 is $eastbount(X) \leftarrow has\_carriage(X, Y), closed(Y), short(Y)$. [3] This target concept is absent in an hypothesis space defined by a most-specific clause constructed with $i = 1$ because the predicates *closed/1* and *short/1* will not occur in a most-specific clause at $i = 1$. Both these predicates are at a variable depth of 2 (i.e. $i = 2$) as they require an input variable of type *carriage* which is itself only introduced at $i = 1$ (through predicate *has_carriage* which requires an input variable of type *train* introduced at i=0, i.e. directly by the head of the clause).

2.2.5 Inverse Entailment and Progol

A landmark ILP system is Progol [Mug95b] which inspired later systems such as Aleph[Sri07]. Progol's ideas are still at the core of most ILP systems today.

Mode-Direct Inverse Entailment (MDIE) was introduced in [Mug95b] as the basis for Progol. The input to an MDIE system is the tuple $S_{MDIE} = \langle M, B, E \rangle$ where M is a set of mode statements, B is a logic program representing the background knowledge and E is a set of examples.

M can be viewed as a set of metalogical statements used to define the hypothesis language $\mathcal{L}_M$. The aim of the system is to find a set of hypothesized clauses H such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that $B, h \models e$ holds. Furthermore, the clauses in H should not cover negative examples.

For any B, h, e this is equivalent to: $B, \neg e \models \neg h$. This form allows hypotheses to be derived

[3] This clause translates to English as: *A train is travelling eastbound if it has a carriage that is both closed and short.* Note that all positive examples are covered by this rule and none of the negative examples are.

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

Having selected an example e and constructed $\bot_e$ Progol conducts a refinement graph search which considers hypotheses h in the interval: $\square \succ h \succeq \bot_e$, where "$\succeq$" denotes θ-subsumption (Definition 5.1). This refinement search is at the core of Progol's cover-set algorithm which is presented in Algorithm A.1 of Appendix A.

Progol's relatively simple cover-set algorithm implements the normal ILP setting but the search lattice defined by the $\bot_e$ clause may be too large, thus only a small fraction of the lattice can be searched. The lattice is searched with the A* algorithm with compression as the admissible heuristic.

Progol's cover-set algorithm is sensitive to the order of the examples and takes local greedy decisions as to which clauses should belong to the induced theory. This dependence on the example order is problematic in some applications. See Section 6.3.1 for an example.

The ILP systems implemented in GILPS have the option to generate the theory at the end, after hypotheses from all examples have been generated. This approach is less greedy but incurs a higher computational cost. See Section 6.3 for more information.

2.3 Overview of feature-based machine learning

Support vector machines (SVMs) [Vap95] and propositional decision trees, such as [Qui93], are two widely used feature-based learners.

SVMs are a statistical machine learning algorithm that can be used for both classification and numerical regression. Let us consider the problem of binary classification with an SVM. In an SVM each problem instance (i.e. an example) is represented as a vector of real numbers. An SVM works by mapping, through a transformation function (the kernel), each of these input vectors into a higher dimensional space.

In this high dimensional space the hyperplane with the largest distance to the nearest training examples of both classes is found. This is called the maximum-margin hyperplane as it maximizes the distance from the closest training examples of both classes. It is equidistant from those training examples.

These training examples, i.e. the input vectors that lie on either side of the equidistant margins, are called the support vectors. The kernel function is a crucial part in an SVM algorithm. Intuitively, the kernel is a function that computes how similar two instances are. This similarity measure is a real number between 0 and 1. Typical kernel functions are: linear, polynomial and radial basis functions. All of these functions have parameters that may require some tuning. An active area of research is the development of kernels for specific data types (e.g. [MRSV06] for 3D molecular structures).

An SVM model is the set of support vectors that lie on the separating hyperplanes. This information is meaningless to a human, making SVMs a black-box classifier. Nevertheless, SVMs are very useful when model understanding is not important but accuracy is paramount.

Decision trees are a popular and mature classification algorithm developed mainly in the mid-1980s [Qui86] with roots dating back to the first computerized induction systems [HMS66]. Standard decision tree algorithms accept features of both real and ordinal types but the target attribute has to be of an ordinal type, thus decision trees are unsuitable for numerical regression.

A decision tree is built using a recursive greedy algorithm that at each step splits the data (on a given attribute) in order to maximize the information gain ratio. To minimize the overfitting of the generated tree a pruning stage is often needed where the resulting tree is simpler and more general.

A decision tree can be described as follows. At each node, starting at the root node, some attribute is being tested. If it has a certain value a branch of the tree is followed, if not another branch is. This is done recursively until a leaf node is reached. At a leaf node, the target attribute prediction is made. It is straightforward to convert a decision tree to a set of (propositional) if-then rules. This set of if-then rules is often easier to interpret than the

original decision tree.

Decision tree rules have some similarity with ILP rules. The main difference is that decision tree rules are propositional (i.e. no quantifiers or variables) and thus less expressive. ILP rules are first-order logic Horn clauses and thus are more expressive as these can encode arbitrary relations between objects.

Machine learning algorithms have been quite successful at solving problems from a broad range of areas [Mit97] such as: search engines, bioinformatics, drug design, fraud detection, speech and handwriting recognition and object recognition in computer vision.

Often the most sought-after feature of a machine learning model is good predictive power (i.e. high accuracy on unseen examples). However, for some applications, like medical diagnosis or drug design, it is critical that the reasoning behind the classification is elicited. In respect to the understandability of the models, learning algorithms can be divided into two major groups: 1) those that generate human-incomprehensible (i.e. black box) models and 2) those that generate human-readable models, often close to natural language. In the first group are support vector machines, in the second are decision trees and ILP.

In contrast to ILP, where examples are described by a logic program, both decision trees and support vector machines require examples to be described as a vector of attributes. As we shall see in Section 2.3.2, there are applications where this vector-of-attributes approach to describe examples presents serious limitations to the ability of expressing the problem to be solved.

2.3.1 ILP versus feature-based learners

ILP and feature-based learners fill different needs. ILP is suited to relational problems and feature-based machine learning to propositional problems. For instance, if the problem to be solved is adequately represented by an example being a vector of independent attributes then a decision tree can be applied with results identical to those of an ILP system, with the advantage of requiring fewer computational resources. Also, if the task to be solved is essentially numerical

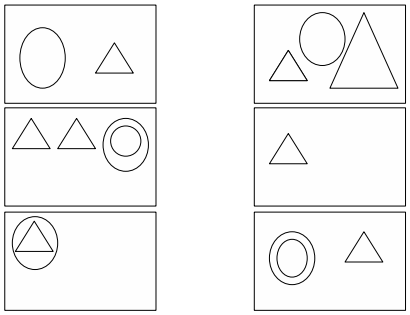


Figure 2.9: Bongard figures

regression, where both features and target are real numbers, ILP is ill-suited. In this scenario a support vector machine would be a sensible choice.

ILP's strength lies in problems requiring relational information. These are problems where examples cannot be easily described as a vector of attributes, as illustrated in the next section. More importantly, many interesting learning problems, e.g. any requiring recursion, cannot be framed at all with a propositional learner. See, for instance, [STNM09] where the Fibonnaci series and the Binomial coefficient function are learned.

2.3.2 Feature-based learning limitations

Propositional machine learning algorithms (e.g. decision trees, neural networks, support vector machines) require examples to be described as a vector of attributes. This simplistic approach can lead to serious difficulties in example representation, as is clearly demonstrated by the class of Bongard problems [Bon70].

The goal of a Bongard problem is to classify two-dimensional scenes consisting of sets of nested polygons (e.g. triangles, rectangles, ellipses). Figure 2.9 illustrates one such problem.

If the relative order, type and inside status of the polygons inside the main rectangle are important to elicit the target concept then it would be extremely cumbersome to describe these six examples using simply a vector of attributes. The first problem is that, since examples have different numbers of polygons, one would have to consider the vector of attributes for all examples to be of the size of the largest example. For the smaller examples, the extra attributes

would be left empty. A more serious problem is that for any arbitrary property (e.g. is polygon A inside polygon B?) considered relevant, there is an explosion of attributes. Moreover, in general, one would not know in advance which properties would be relevant.

In contrast, with a first-order logic representation it is natural to represent an arbitrary example. For instance, the upper left example could be represented by the following Prolog program: $\{rectangle(pol1).\ circle(pol2).\ triangle(pol3).\ contains(pol1, pol2).\ contains(pol3, pol1).\}$.

With ILP one does not need to specify all possible properties in advance. As long as these properties can be constructed from predicates in the mode declarations, they will lie somewhere in the hypothesis lattice and will eventually be found, provided that enough resources are given to the search.

King-Rook vs King legality

Here we present the King-Rook vs King [BM94] legality problem. This problem provides a clear contrast between the limitations of feature-based machine learning and the advantages of ILP.

Consider all chess boards where the only pieces on the board are a white king, a white rook and a black king, and it is white's turn to play. Some of these boards represent legal chess positions and some do not. For instance, any board where the black player is in check is illegal because after its move a player cannot remain in check. Also, any board where more than one piece share the same square is illegal.

The King-Rook vs King problem is intended to determine, given a chess board with only these three pieces, whether the board is legal or illegal. A piece position is described by two integers (row and column) in the range of 1..8. The board is then described by six integers in any pre-arranged order.[4]

This toy problem has the advantage of having a well-defined example distribution, being possible

[4]We will use the order: white rook row, white rook column, white king row, white king column, black king row, black king column.

to build all the training examples and be noise free. There are exactly $(8*8)^3 = 262,144$ unique chess boards for our problem and we can easily determine the legal status for any one of them. Of these 262,144 boards, 175,392 (negative examples) are legal and 86,752 are illegal (positive examples). We consider the target concept to be *illegal* rather than *legal* because it is easier to model illegality.

Despite the apparent simplicity, the correct target definition for this problem (i.e. board illegality) is not as trivial as it may look at first. There are difficult-to-represent cases. For example: *A chess board where the white rook and black king are on the same line (i.e. row or column) may be legal if the white king is in between, blocking the rook.*

The purpose of the learning algorithm in this problem is to build a classification model to predict the legality of unseen chess boards, given chess board representations with the three pieces mentioned (i.e. 6 integers) and the board's legal status (i.e. the target concept). No further attributes were given to the classifiers and no background knowledge was provided to TopLog.

For the next set of experiments we will use a decision tree learner, a support vector machine and an ILP system. The decision tree learner is C5.0, the successor of the well-known C4.5 [Qui93]. The support vector machine we use is LibSVM [CL01], a well-known SVM package. The ILP system used is TopLog, which is presented in Chapter 3. Other ILP systems such as Aleph, Progol or ProGolem would have yielded identical results.

Each classifier was executed with default settings and three models were built with an increasing number of training examples: 0.1% (262 instances), 1% (2,621 instances) and 10% (26,214 instances) of the example space. Ten-fold cross-validation was performed with the training data. The reported results are the average accuracy on the ten folds and the respective standard deviation. Accuracy is defined as the number of correct predictions over the total number of predictions. The default classifier, which says that all boards are legal, has an accuracy of 66.91%. Table 2.1 presents the results.

TopLog could not fully learn (i.e. with 100% accuracy) the target concept because the provided

Algorithm	0.1% train	1% train	10% train
C5.0	$67.5\% \pm 2.9\%$	$70.7\% \pm 0.8\%$	$85.4\% \pm 0.9\%$
LibSVM	$67.8\% \pm 3.7\%$	$74.9\% \pm 0.8\%$	$82.6\% \pm 0.3\%$
TopLog	$91.0\% \pm 7.7\%$	$91.9\% \pm 1.1\%$	$91.8\% \pm 0.5\%$

Table 2.1: Model accuracy for Decision trees, SVMs and ILP on the King-Rook vs King legality problem

```
Rule #1, PosScore=2898 (2898 new), NegScore=50 (50 new)
Accuracy = 98.3% Coverage = 36.8%

illegal(A,_,_,_,A,_).

Rule #2, PosScore=2877 (2524 new), NegScore=60 (60 new)
Accuracy = 98.0% Coverage = 36.6%

illegal(_,A,_,_,_,A).

Rule #3, PosScore=399 (301 new), NegScore=0 (0 new)
Accuracy = 100% Coverage = 5.1%

illegal(A,B,A,B,_,_).

Rule #4, PosScore=389 (301 new), NegScore=0 (0 new)
Accuracy = 100% Coverage = 4.9%

illegal(_,_,A,B,A,B).
```

Figure 2.10: TopLog model for Rook-King vs King problem

background knowledge lacked the predicates that would allow the subtle cases mentioned above to be distinguished.

Nevertheless, even with few training examples, TopLog built the most accurate possible model given that no background knowledge was provided. Figure 2.10 shows the four rules TopLog discovered.

The first rule translates as "whenever the white rook and the black king are in the same row it is an illegal board". The second rule is similar but is for the column rather than the row. The third rule translates as "whenever the white pieces are in the same square the board is illegal". The fourth rule translates as "whenever the rooks are in the same square the board is illegal".

When we contrast the simplicity of these rules with the rules the decision tree generates, the advantage of ILP is clear (even not taking accuracy into account). C5.0 generates on average 75 rules (!) and the rules give almost no insight into what the underlying concept is. For instance, one of the rules C5.0 generated was *if (white rook column>2) and (white king row>6) and (white king column>2) and (black king row>6) and (black king column>3) then illegal board.*

The problem with propositional rules is that they cannot capture concepts that relate two features (e.g. row should be the same). This leads to an explosion of rules and poor generalization. If new examples are taken from a 16x16 board, the decision tree rules would behave quite poorly, whereas the ILP rules would be as accurate as in the 8x8 board.

Support vector machines without a specialized kernel have the same difficulties as decision trees. That is, they generate too many support vectors (the SVM equivalent of a rule), and do not generalize well in this problem. Furthermore, the model an SVM generates is not comprehensible to humans.

Chapter 3

TopLog: ILP with a declarative search bias

In this chapter we introduce a new approach to provide a declarative bias, called Top-Directed Hypothesis Derivation (TDHD), and the ILP system implementing TDHD, TopLog. The initial idea for the TDHD framework is due to Stephen Muggleton and the theoretical foundations for it are the authorship of Stephen Muggleton and Alireza Tamaddoni-Nezhad [MSTN08]. The contribution of the author of this thesis was the creation of TopLog, the ILP system which implements the TDHD framework.

This chapter is based on [MSTN08]. It is arranged as follows. In Section 3.1 we introduce the topic of declarative bias and contextualize TDHD with respect to related work on the field. The theoretical framework for TDHD is introduced in Section 3.2 together with a worked example. In Section 3.3 the ILP system TopLog is presented. Experiments comparing the performance of TopLog and Aleph are given in Section 3.4. We conclude in Section 3.5, discussing TopLog's limitations and possible future work.

3.1 Introduction

TDHD is an approach to provide a declarative search bias that extends the use of the $\bot$ clause in Mode-Directed Inverse Entailment (MDIE) [Mug95b]. In Inverse Entailment $\bot$ is constructed for a single, arbitrarily chosen training example. Refinement graph search is then constrained by the requirement that all hypothesized clauses considered must subsume $\bot$.

In TDHD we further restrict the search associated with each training example by requiring that each hypothesized clause must also be entailed by a given logic program, $\top$. $\top$ can be viewed as a form of first-order declarative bias which defines the hypothesis space, since each hypothesized clause must be derivable from $\top$. The use of the $\top$ theory in TopLog is in some ways comparable to grammar-based declarative biases [Coh94, DT95].

In a context-free grammar it is usual to differentiate between *terminal* and *non-terminal* symbols, where *terminal* symbols are those which can appear in a sentence generated by the grammar. Likewise in $\top$ it is useful to distinguish between *terminal* and *non-terminal* predicates. *Terminal* predicates represent those that can appear in hypotheses derived from $\top$. *Non-terminal* predicates are used for control purposes within $\top$ and cannot appear in hypotheses. However, compared with a grammar-based declarative bias, $\top$ has all the expressive power of a logic program, and can be efficiently reasoned with using logic programming techniques.

3.2 Theoretical framework

We assume the reader to be familiar with the concepts of SLD-derivation and SLD-refutation from Logic Programming. These concepts were presented in Section 2.1 of Chapter 2.

3.2.1 Top-Directed Hypothesis Derivation

The input to a TDHD system is the tuple $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT is a set of "non-terminal" predicate symbols, $\top$ is a logic program representing the declarative bias over

the hypothesis space, B is a logic program representing the background knowledge and E is a set of examples. The following two conditions hold for clauses in $\top$: (a) each clause in $\top$ must contain at least one occurrence of an element of NT while clauses in B and E must not contain any occurrences of elements of NT and (b) clauses in B cannot call clauses in $\top$, i.e. any predicate appearing in the head of some clause in $\top$ must not occur in the body of any clause in B. The aim of a TDHD system is to find a set of consistent hypothesized clauses H, containing no occurrence of NT, such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that the following holds:

$$\top \models h \tag{3.1}$$

$$B, h \models e \tag{3.2}$$

Given the assumptions above we can now show the following lemma.

Lemma 3.1. Example derivability [MSTN08]. *Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ assumptions (3.1) and (3.2) hold only if for each positive example $e \in E$ there exists an SLD-refutation R of $\neg e$ from $\top, B$.*

Using the proof re-ordering lemma, the results of the example derivability lemma can be used to extract implicit hypotheses from the SLD-refutations of a positive example $e \in E$.

Lemma 3.2. Proof re-ordering [MSTN08]. *Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ and a positive example $e \in E$, each SLD-refutation R of $\neg e$ from $\top, B$ can be re-ordered to give $R' = D_h R_e$ where D_h is an SLD-derivation of a hypothesis h for which (3.1) and (3.2) hold.*

Let us now consider a simple example of the proof re-ordering lemma.

Example 3.3. *Let $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT, B , e and $\top$ are defined as follows:*

$$NT = \{\$body\}$$
$$B = \{b_1\} = \{pet(lassie) \leftarrow\}$$
$$E = \{e\} = \{nice(lassie) \leftarrow\}$$
$$\top = \begin{cases} \top_1 : nice(X) \leftarrow \$body(X) \\ \top_2 : \$body(X) \leftarrow pet(X) \\ \top_3 : \$body(X) \leftarrow friend(X) \end{cases}$$

Figure 3.1 shows the linear refutation $R = \langle \neg e, \top_1, \top_2, b_1 \rangle$.

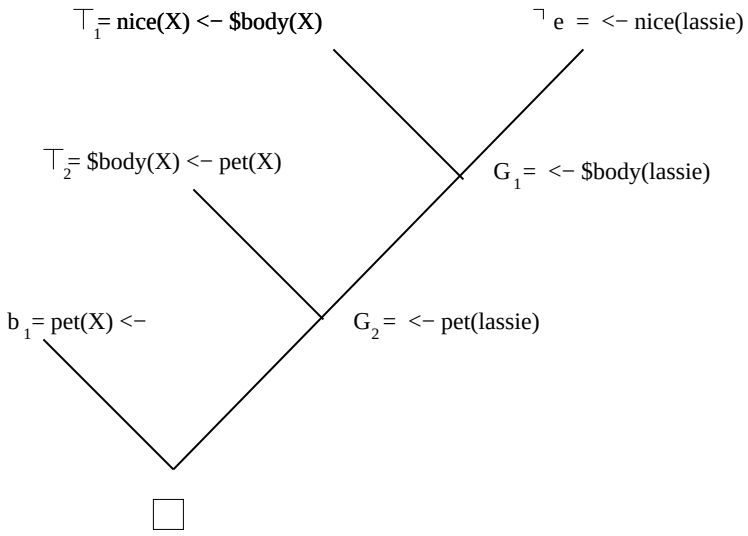


Figure 3.1: SLD-refutation of $\neg e$

We now construct the re-ordered refutation $R' = D_h R_e$ *where* $D_h = \langle \top_1, \top_2 \rangle$ *which derives the clause* $h = nice(X) \leftarrow pet(X)$ *for which (3.1) and (3.2) hold.*

3.2.2 Related work

The SPECTRE ILP system [BIA94] employs an approach related to the use of $\top$. SPECTRE also relies on an overly general logic program as a starting point. However, unlike the TopLog system described in this chapter, SPECTRE proceeds by successively unfolding clauses in the initial theory. The explicit distinction between $\top$ and first-order background knowledge allows TopLog to avoid some of the problems early versions of SPECTRE encountered in learning recursive programs.

TDHD is also related to Explanation-Based Generalization (EBG) [KCM87]. EBG starts with an initial general theory, which is used to explain individual examples. EBG distinguishes between *operational* and *non-operational* predicates in a similar fashion to the terminal and non-terminal distinction made in TDHD. As with TDHD, the proofs of individual examples in EBG are generalized to provide clausal explanations by dropping all non-operational predicates.

However, like SPECTRE, EBG does not make the key TDHD distinction between the $\top$ theory and background knowledge. Thus in EBG the initial starting theory is interpreted as domain

knowledge, which means that the explanations can be assumed correct by derivation. For this reason EBG and the associated EBL [DM86] were viewed as a form of deductive learning, aimed at speeding up program performance. By contrast, TDHD is a form of inductive learning which uses the $\top$ theory as a declarative bias in order to define the hypothesis space.

3.3 TopLog

TopLog is the ILP system developed to implement the TDHD framework. It is one of the ILP systems implemented in GILPS [San10]. See Section 6.4 for a tutorial on how to use GILPS, and in particular how to run GILPS in TopLog mode.

3.3.1 Top theory construction from mode declarations

As the user of TopLog may not be familiar with specifying a search bias in the form of a logic program, TopLog is able to create a $\top$ theory automatically from traditional user-specified mode declarations. In this way input compatibility with existing ILP systems is ensured. However, the user of TopLog should write his own $\top$ theory specific to the learning task. This user-specified $\top$ theory should incorporate enough domain knowledge to restrict the hypothesis space better than the generic $\top$ theory built automatically from the mode declarations.

Figure 3.2 shows a simplified example of mode declarations and the respective, automatically constructed, $\top$ theory. This $\top$ theory is extremely simplified as it disregards the input/output mode of the arguments of the mode declarations predicates and the respective variable types. Nevertheless, this simplified example is useful as it shows the parallelism (and contrast) between mode declarations and $\top$.

modeh(mammal(+animal)).
modeb(has_milk(+animal)).
modeb(has_eggs(+animal)).

$$\top = \begin{cases} \top_1 : \text{mammal}(X) \leftarrow \$\text{body}(X) \\ \top_2 : \$\text{body}(X) \leftarrow \\ \top_3 : \$\text{body}(X) \leftarrow \text{has\_milk}(X), \$\text{body}(X) \\ \top_4 : \$\text{body}(X) \leftarrow \text{has\_eggs}(X), \$\text{body}(X) \end{cases}$$

Figure 3.2: Mode declarations and a $\top$ theory automatically constructed from it

We now provide a detailed account, via a second example, of the actual process by which the mode declarations are actually compiled to a $\top$ theory. This description may be useful to guide users in developing specific $\top$ theories for their domains.

Like a grammar, $\top$ has *terminal* and *non-terminal* predicate symbols. The *non-terminal* predicates of $\top$ are control predicates and do not appear in a hypothesis. The *terminal* predicates appear in the hypothesis and are defined in the background knowledge.

When translating the mode declarations to a $\top$ theory, the part of $\top$ corresponding to the *non-terminal* predicates is fixed irrespective of the particular mode declarations. The part of $\top$ responsible for introducing the *terminal* predicates (i.e. the literals appearing in a valid hypothesis) is compiled from the mode declarations and thus vary.

Immutable part of $\top$

Each clause in $\top$ has the form *theory(+ClauseID, +Head, +Body)*, where *ClauseID* is a positive integer uniquely identifying each predicate in $\top$. The list of all clause identifiers used in the proof of an example is later needed to derive a hypothesis. See Section 3.3.2 for further details on hypothesis derivation. The second argument, *Head*, is the head of a $\top$ theory clause and *Body* is the body of the same clause, as a list of literals. The *non-terminal* predicates of $\top$ are presented in Figure 3.3.

The resemblance between $\top$ and a grammar is notorious. The *body* predicate (i.e. ids=11 and 12) is the control predicate responsible for either terminating the clause derivation (id=11) or specializing the clause further by introducing a compatible literal (id=12). In this latter case the body predicate recurses to repeat the decision in the next iteration. The *atom* predicate call in *body* predicate with id=12 will introduce the *terminal* symbols of $\top$.

Two lists are ubiquitous in $\top$: Input Terms (*InTerms*) and Output Terms (*OutTerms*). Input terms are terms that may be used later as input arguments to literals of subsequent clause specializations. OutTerms are output terms from the target predicate head which are not yet instantiated but will become instantiated as $\top$ unfolds. Each element of a terms list is a tuple

```
% body(+InTerms, +OutTerms)
theory(11, body(_, []), []).
theory(12, body(InTerms, OutTerms),
           [atom(InTerms, OutTerms, NInTerms, NOutTerms),
            body(NInTerms, NOutTerms)]).

% member(+Elem, +List)
theory(20, member(H, [H|_]), []).
theory(21, member(H, [_|T]), [member(H, T)]).

% minus(+InList, +Elem, -OutList)
theory(22, minus([], _, []), []).
theory(23, minus([H|T], H, T), [!]).
theory(24, minus([V|T], H, [V|R]), [minus(T, H, R)]).

% mergeTerms(+Terms, +CurInTerms, +CurOutTerms, -NewInTerms, -NewOutTerms)
theory(30, mergeTerms([], InTerms, OutTerms, InTerms, OutTerms), []).
theory(31, mergeTerms([V|OutVars], CInTerms, COutTerms, NInTerms, NOutTerms),
           [ member(V, CInTerms),!,
             mergeTerms(OutVars, CInTerms, COutTerms, NInTerms, NOutTerms)
           ]).
theory(32, mergeTerms([V|OutVars], CInTerms, COutTerms, NInTerms, NOutTerms),
           [  minus(COutTerms, V, TempOutTerms),
              mergeTerms(OutVars, [V|CInTerms], TempOutTerms, NInTerms, NOutTerms)
           ]).
```

Figure 3.3: Immutable predicates of $\top$

⟨ground term/term type⟩ (e.g. 5/int). A clause is only accepted as valid (by id=11), when the output terms list is empty (i.e. there are no free variables in the head of the clause).

The control predicate *mergeTerms* merges a set of terms with the current Input and Output terms lists and returns new lists. If a term already exists in the *InTerms* it is ignored (id=31). Otherwise (id=32), this term is new and it is added to the new set of *InTerms*. If the term previously existed in *OutTerms* it is removed from the new *OutTerms*, as the term is now available to be an input argument. The predicates *member* and *minus* are needed by *mergeTerms* and are in $\top$ in order for the example proofs to be self-contained.

Problem-dependent part of $\top$

We now introduce the $\top$ theory predicates which are mode declarations dependent. These predicates introduce the *terminal* predicates of $\top$. Consider the mode head and body declarations

of Figure 3.4.

```
:- modeh(active(+drug)).
:- modeb(*, atm(+drug, -atomid, #element)).
:- modeb(*, bond(+drug, +atomid, -atomid, #int)).
```

Figure 3.4: Mode declarations for a typical drug activity problem

Figure 3.5 shows the $\top$ theory resulting from compiling these mode declarations. TopLog performs this compilation without user intervention.

```
theory(1,
         head(active(+drug), active(V)),
         [
          mergeTerms([V/drug], [], [], InTerms, OutTerms),
          body(InTerms, OutTerms)
         ]).

%atom(+InTerms, +OutTerms, -NewInTerms, -NewOutTerms)
theory(100,
       atom(InTerms, OutTerms, NInTerms, NOutTerms),
       [
        member(A/drug, InTerms),
        call_pred(10, atm(+drug, -atomid, #element), atm(A, B, C)),
        mergeTerms([B/atomid], InTerms, OutTerms, NInTerms, NOutTerms)
       ]).
theory(101,
       atom(InTerms, OutTerms, NInTerms, NOutTerms),
       [
        member(A/drug, InTerms),
        member(B/atomid, InTerms),
        call_pred(10, bond(+drug, +atomid, -atomid, #int), bond(A, B, C, D)),
        mergeTerms([C/atomid], InTerms, OutTerms, NInTerms, NOutTerms)
       ]).
```

Figure 3.5: Problem-dependent part of $\top$ resulting from the compilation of the mode declarations

The clause with id=1 is compiled from the modeh declarations and defines the head of the hypothesis. This clause starts by merging any duplicate terms in the example and then calls the *body* predicate with the preprocessed Input and Output terms.

Clauses with ids>= 100 are atoms of the grammar defining the literals which may appear in the body of a hypothesis. In an atom clause we start by instantiating the input variables of the literal to be added. This is accomplished by the *member* predicate calls before *call_pred*.

Note that the input variables are selected from the possible *InTerms* of the same type of the predicate input argument.

After this step, the literal to be added to the body of the current hypothesis is called with its input variables instantiated. This is accomplished by the special *call_pred* which performs a call to the background knowledge up to the recall specified by the first argument.

If *call_pred* succeeds, the variables in the constant and output placeholders get bound. Then *mergeTerms* merges the output terms from the called predicate to the current *InTerms* list and the control is returned to the *body* control predicate.

If *call_pred* fails we have to backtrack to a previous choice point. That may be a short backtrack to have a different binding for the input variables or a larger backtrack which may remove recently added body literals.

3.3.2 Hypothesis derivation

The hypothesis derivation process has three distinct steps. In the first step, an example, e, is proved from the background knowledge and $\top$. The $\top$ theory is resolved away, having e as its start clause (i.e. matched with $\top_1$). Each successful derivation yields a ground clause entailed by $\top$. Along with each derivation we also have its proof, the sequence of clause identifiers from $\top$ that were resolved.

For instance, using the $\top$ theory from Figure 3.2 and $B = \{b_1\} = \{\text{has\_milk(dog)}\}$ to derive refutations for example $e = \text{mammal(dog)}$, the following two refutations would be yielded: $r_1 = \langle \neg e, \top_1, \top_2 \rangle$ and $r_2 = \langle \neg e, \top_1, \top_3, b_1, \top_2 \rangle$.

In the second step, the proof is re-ordered according to Lemma 3.2. Applying Lemma 3.2 to r_1 and r_2 yields, respectively, the clauses $c_1 = mammal(dog)$ from $\langle \top_1, \top_2 \rangle$ and $c_2 = mammal(dog) \leftarrow has\_milk(dog)$ from $\langle \top_1, \top_3, \top_2 \rangle$.

In the third step, hypotheses are finally generated by performing a least general variablization on the clauses returned. In a least general variablization, each unique ground term is assigned

a distinct variable. For instance, applying least general variablization to $c_1 = mammal(dog)$ and $c_2 = mammal(dog) \leftarrow has\_milk(dog)$ yields, respectively, the clauses $h_1 = mammal(X)$ and $c_2 = mammal(X) \leftarrow has\_milk(X)$.

3.3.3 Cover-set algorithm

Algorithm 3.1 shows the pseudo-code for the TopLog cover-set algorithm.

Algorithm 3.1 TopLog's cover-set

Input: Examples E, top theory $\top$, background knowledge B
Output: Theory T, a set of definite clauses

```
 1: Let T = {}
 2: Let E+ = all positive examples in E
 3: Let ⊤ = Top theory created from the mode declarations or specified by the user
 4: while E+ contains unseen positive examples do
 5:    Let e = first unseen positive example from E+
 6:    Mark e as seen
 7:    Let H = all hypotheses derived when the start clause of ⊤ = e
 8:    Let Ce = highest scoring clause in H
 9:    if Ce has positive score then
10:       T := T ∪ Ce
11:       Ec+ := all positive examples clause Ce covers
12:       E+ := E+ \ Ec+
13:    end if
14: end while
15: return T
```

Notice that this TopLog cover-set algorithm is very similar to the Progol (and Aleph) algorithm presented in Algorithm A.1. The difference is in line 7 where all hypotheses for e are derived from $\top$ instead of a A* search in the lattice bounded by $\bot_e$. The scoring function used in line 8 may be user selected (see Section 6.4.3 for a list of possible scoring functions). By default the scoring function is compression.

TopLog can also be executed in a global theory construction mode where hypotheses from all the examples are initially generated and only then the induced theory is constructed. In this mode, hypotheses generation and theory construction are independent. No construction of the theory takes place while generating hypotheses and no positive examples are removed (as occurs in line 11 of Algorithm 3.1) during the search. The global theory construction mode is further

detailed in Section 6.3 and was employed in [MSTN08]. Note, however, that the feature that distinguishes TopLog from other ILP systems is the hypothesis derivation process guided by a $\top$ theory.

3.3.4 Comparison with Aleph

Mode Directed Inverse Entailment ILP systems like Aleph [Sri07] construct firstly a most-specific clause, $\bot$, to bound the hypothesis space and then perform an A* search in this lattice. Valid hypotheses are subsets of $\bot$ (i.e. contain only a subset of the literals in $\bot$).

TopLog does not require the construction of $\bot$, instead having its search constrained by $\top$. The hypotheses TopLog derives are derivations of $\top$ but with no guiding heuristic. There is an advantage in using a $\top$ theory when one knows beforehand that the hypothesis must have a particular format. In this case, one can write a problem-specific $\top$, constraining the hypothesis search space better than with mode declarations.

Unfortunately, writing a custom $\top$ theory requires both problem-specific knowledge and a better understanding of logic and ILP than is needed when using mode declarations.

3.4 Experimental evaluation

In this section we empirically compare TopLog and Aleph performance on four well-known ILP datasets.

3.4.1 Materials

The datasets chosen were: mutagenesis [KMSS96], carcinogenesis [SMS97], alzheimers-amine [KSS95] and DSSTox [RW00] mainly because they are well-known to the ILP community and are good examples of real-world problems where relational knowledge is important.

In these datasets the purpose is to characterize an active molecule where the definition of molecular activity is problem-specific. The ILP system is given examples of molecules that are active (i.e. positives) and examples of molecules that are not active (i.e. negatives). The task is then to induce a theory that entails as many of the positive examples as possible while entailing as few of the negative examples as possible.

To ensure the problems are interesting from an ILP perspective we have only used structural features (e.g. atoms, bonds, and structural motifs formed of atoms and bonds). When we use quantitative features (e.g. logp and lumo) the predictive accuracies are higher but the hypotheses found are less insightful in the sense that they offer fewer clues on how to build such molecules. Table 3.1 summarizes the dataset information.

Dataset	$\#E^+$	$\#E^-$	# E	Def. Acc.	#Modeb	#Background
Mutagenesis	125	63	188	66.5%	4	14,379
Carcinogenesis	162	136	298	54.4%	40	24,672
Alzheimers	343	343	686	50.0%	32	628
DSSTox	220	356	576	61.8%	2	27,793

Table 3.1: Dataset statistics

The meaning of the columns is: number of positive examples, number of negative examples, total number of examples, default accuracy, number of body mode declarations and size of the background knowledge measured in number of clauses. The default accuracy is the accuracy yielded by the simple model that classifies an example as belonging to the most common class; its value is thus $max(\#E^+, \#E^-)/\#E$ and should be considered the baseline that any classifier should improve upon.

3.4.2 Methods

We chose to compare TopLog against Aleph [Sri07] because Aleph is a Mode Directed Inverse Entailment ILP system and is also implemented in YAP Prolog.

The experiments were performed on an Intel Core 2 Duo @ 2.13 GHz with 2GB of RAM on Linux. Aleph [Sri07] version 5.0 was used. TopLog is one of the ILP systems in GILPS [San10].

GILPS version 0.16 was used. The four datasets are freely available from GILPS's webpage. Both TopLog and Aleph were executed on YAP 6.0.6 [Cos09].

In order to ensure a fair test, Aleph and TopLog were executed with settings as similar as possible. The settings used for both systems on all datasets were: clause length (i.e. maximum number of literals in the body of a hypothesis) = 4, cross-validation folds = 10, maximum number of negative examples allowed to be covered by an hypothesis = 20, minimum precision of an hypothesis = 70%, recall = 10, scoring function = compression, maximum number of nodes (i.e. hypotheses) generated per example = 1000, theory construction mode = incremental, i.e. retract positive examples covered while constructing the theory. This corresponds to "induce" in Aleph. A thorough description of these settings is provided in Section 6.4.3

Note that no custom $\top$ theory has been developed for TopLog. TopLog created the $\top$ theory for each dataset by the mechanism explained in Section 3.3.1.

3.4.3 Results and Discussion

In Table 3.2 the results of the experiments are presented. The CV-accuracy column is the average (over the ten folds) percentage of correct predictions made by the ILP model with the respective standard deviation on the left-out test fold. The time column is the total running time, in CPU seconds, that the ILP system took to build the 10-fold cross-validated theory.

	Aleph		TopLog	
Dataset	CV accuracy	Time(s)	CV accuracy	Time(s)
Mutagenesis	78.9%±11%	26	80.0%±9.5%	52
Carcinogenesis	60.2%±8.7%	76	57.1%±14.4%	269
Alz-amine	74.2%±3.9%	140	67.7%±5.6%	988
DSSTox	70.2%±6.4%	218	68.1%±2.9%	117

Table 3.2: CV-accuracy and running time comparison between Aleph and TopLog

While the cross-validated accuracies of TopLog and Aleph are not statistically significantly different (with a t-test at p=0.05), the running times of TopLog are, in three of the four datasets, considerably longer than Aleph. The main reason for this difference in running times

is that TopLog's hypothesis search is blind whereas Aleph, like Progol, employs an A* search guided by compression.

For a given example, TopLog derives all possible hypotheses from the $\top$ theory until either no more hypotheses are derivable or the maximum number of hypotheses that are allowed to be derived has been reached. Only then, from these hypotheses, does TopLog pick the one that has higher compression.

In contrast, while Aleph is searching the lattice of hypotheses defined by the most-specific clause of one example, its search process is guided by the compression heuristic, which allows it to prune large parts of the hypothesis space. Therefore, Aleph may consider fewer clauses than TopLog or consider different sets of clauses.

In [MSTN08] we compared TopLog with Aleph in the same datasets; however, TopLog's running times were, then, significantly smaller. The main reason for this difference is that in [MSTN08] we emphasized two features of TopLog, efficient cross-validation and global theory construction, above the Top Directed Hypothesis Derivation.

The initial TopLog system presented in [MSTN08] did not have the option to do the incremental theory construction that Aleph and Progol perform. In the present empirical evaluation we decided not to use efficient cross-validation nor global theory construction (see Section 6.3) so that TopLog's setting would be as similar as possible to Aleph, varying only how hypotheses are derived.

3.5 Conclusions and future work

The key innovation of the TDHD framework is the introduction of a first-order $\top$ theory which constrains the hypothesis search space in a top-down fashion. SLD-resolution is then used to derive hypotheses from $\top$. Unlike other forms of declarative bias, in TDHD the $\top$ theory is a logic program, allowing it first-class status for logic-program-based reasoning mechanisms. For instance, in line with recent interest in learning declarative bias [BT07], $\top$ could itself

potentially be learned using ILP techniques.

The TDHD framework was implemented in a new general ILP system, TopLog. An empirical comparison demonstrates that while TopLog is competitive in predictive accuracy with Aleph, it underperforms Aleph in running times. This underperformance, as explained, is mainly due to a blind search of the hypothesis space.

We illustrated how TopLog converts from traditional mode declarations to a $\top$ theory automatically. However, with the present TopLog, the benefits of the TDHD can only be reaped if a custom $\top$ theory is specified. Otherwise, when the automated $\top$ theory constructor based on the mode declarations is used, the hypotheses space is equivalent to the one defined by a set of mode declarations.

A promising area to extend TopLog is to upgrade $\top$ from a regular logic program to a stochastic logic program (SLP) [Mug95a] where each clause in $\top$ is labelled with a probability. These probabilities would have an initial prior and would be updated during the hypotheses derivation process to reflect the areas of the hypothesis space that looked most promising. This SLP extension of $\top$ would allow for a stochastic sampling of the hypothesis space, biased towards yielding high-compression clauses, solving the blindness of TopLog's hypotheses search.

Chapter 4

ProGolem: An efficient bottom-up ILP learner

In this chapter we introduce ProGolem, a new efficient bottom-up ILP learner capable of learning complex non-determinate concepts (i.e. target predicates). The chapter is arranged as follows. In Section 4.1 we contextualize and explain the need for ProGolem. In Section 4.2 the concept of asymmetric relative minimal generalization (*armg*) is introduced. In Section 4.3 the main algorithms and control strategy of ProGolem are explained and a comparison with other ILP systems is made. A thorough empirical evaluation of ProGolem on a representative set of real-world and artificial datasets is presented in Section 4.4. We conclude and discuss future work in Section 4.5.

This chapter is based on [MSTN09], with more detail on ProGolem's algorithms and empirical evaluation but less detail on the theoretical framework. In addition, the empirical evaluation section contains further experiments from [SM10b].

The initial idea for ProGolem is due to Stephen Muggleton and the theoretical framework for it, namely the concept of *armg*, is the authorship of Stephen Muggleton and Alireza Tamaddoni-Nezhad. The contribution of the author of this thesis was the development of ProGolem along with the theorems and algorithms presented in Section 4.3. All experiments detailed in Section 4.4 and respective analysis are also the author's.

4.1 Introduction

There are two key tasks at the heart of ILP systems: 1) enumeration of clauses which explain one or more of the positive examples and 2) evaluation of the numbers of positive and negative examples covered by these clauses. Top-down refinement techniques, such as those found in [Sha83, Qui90, RB93], use a generate-and-test approach to problems 1) and 2). A new clause is first generated by application of a refinement step and then tested for coverage of positive and negative examples.

It has long been appreciated in AI [Nil80] that generate-and-test procedures are less efficient than ones based on test-incorporation. The use of the most-specific clause in Progol [Mug95b] represents a limited form of test-incorporation in which, by construction, all clauses in a refinement graph search are guaranteed to cover at least the example associated with the most-specific clause.

The use of relative least general generalization ($rlgg$) in Golem [MF92] provides an extended form of test-incorporation in which constructed clauses are guaranteed to cover a given set of positive examples. However, in order to guarantee polynomial-time construction the form of $rlgg$ in Golem was constrained to determinate clauses. Without this constraint Plotkin [Plo71] showed that the length of $rlgg$ clauses grows exponentially in the number of positive examples covered.

In the present work we explore variants of Plotkin's $rlgg$ which are based on subsumption order relative to a most-specific clause [TNM09]. We introduce the concept of asymmetric relative minimal generalization ($armg$) and show that the length of $armg$s is bounded by the length of the initial most-specific clause. Hence, unlike in Golem, we do not need the determinacy restrictions to guarantee a polynomial bound to $armg$ length.

Our ILP system ProGolem combines the most-specific clause construction of Progol with the bottom-up control strategy of Golem using $armg$s in place of determinate $rlgg$s. Top-down ILP systems such as Progol [Mug95b] and Aleph [Sri07] limit the maximum complexity of learned clauses, due to a search bias which favours short clauses. The small refinement steps in such

systems make them ill-suited to learning long, non-determinate, target concepts.

An advantage of ProGolem over classical top-down ILP systems, as we will show in Section 4.4, lies on learning long, non-determinate, target concepts (i.e. predicates). The size of a predicate is defined by the number of literals in its body. Whereas the target predicate complexity is problem dependent and usually unknown a priori, non-determinate background knowledge is frequent in real-world applications, e.g. [RW00], [SMS97], [CHH+02].

4.2 Theoretical framework

We start by reviewing some of the basic concepts from the ILP systems Golem and Progol in Section 4.2.1. The concept of Asymmetric Relative Minimal Generalization is introduced in Section 4.2.2 where some of its properties are demonstrated.

4.2.1 Preliminaries

We assume the reader to be familiar with the basic concepts from Logic Programming and Inductive Logic Programming. These concepts were presented in Chapter 2.

ProGolem borrows ideas from the ILP systems Golem [MF92] and Progol [Mug95b]. This section is a brief summary of the ILP concepts from Golem. The concepts from Progol, i.e. mode declarations, most-specific clause, and Mode-Directed Inverse Entailment, were presented in Section 2.2.

Proposition 4.1. Subsumption lattice [MSTN09] *Let $\mathcal{C}$ be a clausal language and $\succeq$ be the θ-subsumption order. Then the equivalence classes of clauses in $\mathcal{C}$ and the $\succeq$ order define a lattice. Every pair of clauses C and D in the subsumption lattice have a least upper bound called* least general generalization (lgg)*, denoted by $lgg(C, D)$, and a greatest lower bound called* most general specialization (mgs)*, denoted by $mgs(C, D)$.*

Plotkin investigated the problem of finding the least general generalization (lgg) for clauses

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

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

The ILP system Golem [MF92] is based on Plotkin's notion of *rlgg* of clauses. Golem uses extensional background knowledge (i.e. only ground facts, no clauses) to avoid the problem of non-finite *rlggs*. Extensional background knowledge B can be generated from intensional background knowledge B' by generating all ground unit clauses derivable from B' in at most h resolution steps. The parameter h is provided by the user.

The *rlggs* constructed by Golem were forced to have a tractable number of literals by requiring determinate clauses. A clause is determinate if all its literals are determinate. A predicate is determinate if it admits at most one solution when its input variables are bound. Many real-world applications, including the learning of chemical properties from atom and bond descriptions, require non-determinate background knowledge.

4.2.2 Asymmetric relative minimal generalizations

The construction of the *lgg* of clauses in the general subsumption order is inefficient as the cardinality of the lgg of two clauses can grow very rapidly. For example, with Plotkin's *rlgg*, clause length grows exponentially in the number of examples [Plo71]. Hence, an ILP system like Golem [MF92], which uses *rlgg*, was constrained to determinate clauses to guarantee polynomial-time construction.

Asymmetric relative minimal generalization (*armg*) is a variant of Plotkin's *rlgg* based on

subsumption order relative to a most-specific clause that does not need determinacy restrictions. The notion of subsumption order relative to a most-specific clause, $\succeq_\perp$, and refinement operators for $\succeq_\perp$ were introduced in [TNM09].

An *armg* is based on pairs of positive examples and, as in Golem, is guaranteed to cover all positive examples used to construct it. *Armgs* have the same advantage as *rlgg*s in Golem but, unlike *rlgg*s, the length of an *armg* is bounded by the length of $\perp_e$.

Below, we define asymmetric relative minimal generalization and highlight some of its properties.

Definition 4.2. Asymmetric relative minimal generalization [MSTN09] *Let E be the set of examples, M the mode definitions, B the background knowledge, $\perp_e$ be most-specific clause for e, e and e' be positive examples in E and C' a clause in the hypothesis space defined by M. C' is an asymmetric common generalization of e' and e relative to $\perp_e$, denoted by $C' \in arcg_\perp(e'|e)$, if $C' \succeq_\perp \perp_e$ and $B, C' \models e'$. C is an asymmetric minimal generalization of e' and e relative to $\perp_e$, if $C \in arcg_\perp(e'|e)$ and $C \succeq_\perp C' \in arcg_\perp(e'|e)$ implies C is subsumption-equivalent to C' relative to $\perp_e$.*

Example 4.3. *Let $M = \{modeh(h(+e)), modeb(q(+e,-t)), modeb(r(+e,-t))\}$ be the mode definitions, $B = \{q(a,a),\ r(a,a), q(b,b), q(b,c), r(c,d)\}$ be the background knowledge and $e = h(a)$ and $e' = h(b)$ be two positive examples. Then we have $\perp_e = h(X) \leftarrow q(X,X), r(X,X)$. Clauses $C = h(V_1) \leftarrow q(V_1,V_1)$, $D = h(V_1) \leftarrow q(V_1,V_3), r(V_3,V_5)$ and $E = h(V_1) \leftarrow q(V_1,V_3)$ are all in $arcg_\perp(e'|e)$. Of these, clauses C and D are also in $armg_\perp(e'|e)$.*

As Example 4.3 shows, *armg*s are not unique. More formally:

Lemma 4.4. Armg non-uniqueness [MSTN09] *The set $armg_\perp(e'|e)$ can contain more than one clause which are not subsumption-equivalent relative to $\perp_e$.*

As the name suggests, *armg*s are in general asymmetric, i.e. $armg_\perp(e|e') \neq armg_\perp(e'|e)$. Example 4.5 illustrates the asymmetry.

Example 4.5. *Let* $B = \{a(e1, b1), a(e1, c1), b(b1), c(c1), a(e2, b2), b(b2), d(b2)\}$ *be the background knowledge,* $e = h(e1)$ *and* $e' = h(e2)$ *and the mode declarations:*
$modeh(h(+e)), modeb(a(+e, -t)), modeb(b(+t)), modeb(c(+t)), modeb(d(+t))$.

The variablized most-specific clauses are then: $\bot_e = h(A) \leftarrow a(A, B), a(A, C), b(B), c(C)$ *and* $\bot_{e'} = h(A) \leftarrow a(A, B), b(B), d(B)$. *We then have* $C_1 = h(A) \leftarrow a(A, B), a(A, C), b(B) \in armg_\bot(e'|e)$ *and* $C_2 = h(A) \leftarrow a(A, B), b(B) \in armg_\bot(e|e')$. *However,* $C_1 \notin armg_\bot(e|e')$ *and* $C_2 \notin armg_\bot(e'|e)$.

The length of an *armg* is bounded by the initial length of $\bot$ as an *armg* is a subsequence of $\bot$.

Lemma 4.6. Armg length bound [MSTN09] *For each* $C \in armg_\bot(e'|e)$ *the length of* C *is bounded by the length of* $\bot_e$.

From Theorem 26 in [Mug95b], it follows that the length of $\bot_e$ is polynomially bounded in the number of mode declarations for a fixed value of the maximum variable depth in a most-specific clause (i), and the maximum number of solutions a predicate may yield ($recall$).

Thus, unlike the *rlgg*s of Golem, *armg*s not only do not need determinacy restrictions but also ensure a polynomial bound on the length of clauses generated. *Armg*s handling non-determinate background knowledge is the key for enabling ProGolem to be applied to a wider range of problems.

4.3 ProGolem

ProGolem is the ILP system developed to implement the concept of *armg*. It is one of the ILP systems implemented in GILPS [San10]. See Section 6.4 for a tutorial on how to use GILPS, and in particular how to run GILPS in ProGolem mode.

In this section we guide the reader through ProGolem's main algorithms. We start by explaining the cover-set algorithm of ProGolem which will then lead to the other main algorithms: beam-search iterated armg in Section 4.3.2, armg construction in Section 4.3.3 and negative-based

reduction in Section 4.3.4. We conclude in Section 4.3.5 with a comparison between ProGolem and Aleph.

4.3.1 Cover-set algorithm

ProGolem, as in Golem and Progol, uses the cover-set approach to construct a theory consisting of more than one clause. ProGolem's cover-set algorithm is shown in Algorithm 4.1.

Algorithm 4.1 ProGolem's cover-set

Input: Examples E, background knowledge B, mode declarations M
Output: Theory T, a set of definite clauses

```
1: Let T = {}
2: Let E+ = all positive examples in E
3: while E+ contains unseen positive examples do
4:    Let e = first unseen positive example from E+
5:    Mark e as seen
6:    Let C = Best_armg(e, E, M) (see Section 4.3.2)
7:    Let C' = Negative_based_reduction(C, E) (see Section 4.3.4)
8:    if C_e has positive score then
9:       T := T ∪ C_e
10:      E+_c := all positive examples clause C' covers
11:      E+ := E+ \ E+_c
12:   end if
13: end while
14: return T
```

At each iteration of the cover-set algorithm ProGolem repeatedly constructs clauses using the beam-search iterated armg algorithm (line 6) to select the highest-scoring *armg* with respect to an initial seed example, e (line 4). The beam-search iterated armg algorithm is detailed in Section 4.3.2.

The clauses yielded by the beam-search iterated armg algorithm need to be further generalized. We employ a negative-based reduction algorithm (line 7) to prune literals from the body of C that are non-essential. A non-essential literal is a literal that, if removed, does not change the negative coverage of the clause. The negative-based reduction algorithm is detailed in Section 4.3.4.

The remainder of ProGolem's cover-set algorithm closely resembles TopLog's. All the consid-

erations made in Section 3.3.3 about global theory construction, efficient cross-validation and scoring function also apply to ProGolem. This similarity is because all ILP systems in GILPS have the same structure; the distinguishing factor is how hypotheses are derived.

Note that the intermediate clauses considered by ProGolem will be long and potentially non-determinate. This is a significant issue in ProGolem, which we will deal with in Section 6.2. Golem also considered long clauses but only determinate ones. Determinate clauses are inexpensive to evaluate as the computational complexity is linear in the length of the clause.

4.3.2 Beam-search iterated armg

The beam-search iterated *armg* algorithm pseudo-code is presented in Algorithm 4.2.

Algorithm 4.2 Beam-search iterated armg

Input: Examples E, positive example e, mode declarations M, sample size K, beam-width N
Output: Highest scoring *armg* generated from seed example e

```
 1: Let ⊥_e = most-specific clause(e, B, M) (see Algorithm A.2)
 2: Let best_armgs = {⊥_e}
 3: repeat
 4:    Let best_score = score of the highest scoring clause in best_armgs
 5:    Let E_s = K random selected positive examples from E
 6:    Let new_armgs = {}
 7:    for each clause C ∈ best_armgs do
 8:       for each e' ∈ E_s do
 9:          Let C' = armg(C, e') (see Section 4.3.3)
10:          if score(C) > best_score then
11:             new_armgs := new_armgs ∪ C'
12:          end if
13:       end for
14:    end for
15:    if new_armgs ≠ {} then
16:       best_armgs := highest scoring N clauses from new_armgs
17:    end if
18: until new_armgs = {}
19: return highest scoring clause from best_armgs
```

The goal of the beam-search iterated armg algorithm is to generate the best possible *armg* from a given positive seed example. We start by generating the most-specific clause for e (line 1), $\bot_e$, and set it as our initial best *armg* (line 2). We then randomly select K (sample size) positive

examples (line 5) and perform K *armg*s construction operations (line 9) with the current best *armg*s (only $\perp_e$ yet).

The highest scoring N of these *armg*s are selected as the seed *armg*s for the next iteration (line 16). This process is iterated until we can no longer generate higher scoring *armg*s (line 18). We then return the highest scoring *armg* from the previous iteration.

An *armg* is a definite clause like any other and its score is evaluated with the user-selected scoring function (e.g. compression, accuracy, precision, ...). K and N are the two parameters that control the resources allocated to the beam-search iterated *armg* algorithm. Their values impact significantly the running times and, to a lesser extent, the quality of the generated clause. The default values of N and K in ProGolem are 2 and 10, respectively.

The *armg* construction algorithm of line 9 is presented in the next section.

4.3.3 Armg algorithm

ProGolem's main refinement operator is the *armg*. Algorithm 4.3 presents the pseudo-code of the *armg* construction algorithm of ProGolem.

Algorithm 4.3 *Armg* construction

Input: Clause C, background knowledge B, positive example e
Output: $armg(C, e)$

```
1: if head of C does not match with e then
2:    return □
3: end if
4: Let D = C
5: while D ⊭ e do
6:    Let b_i = first blocking literal of D (see Definition 4.9)
7:    Remove b_i from D
8:    Remove literals from D which are not head-connected
9: end while
10: return D
```

The *armg* construction algorithm receives as input a clause C, a positive example, e, and the background knowledge, B. It outputs a clause D that entails e with respect to B. Clause D has a subset of the literals of C.

The *armg* algorithm works by systematically dropping the first literal, b_i, that is responsible for C not entailing e. After b_i has been removed, it is then necessary to remove the non head-connected literals that may now exist. These b_i literals are called blocking literals, see *First blocking literal* sub-section below for further details.

For a literal to be head-connected all its input variables must be ground before it is called.

Definition 4.7. Head-connected literal *A literal b_i of a definite clause $h \leftarrow b_1, .., b_n$ is said to be head-connected if, and only if, all its input arguments appear either as an input argument of h or in a previous body literal b_j, where $1 \leq j < i$.*

The input/output classification of the arguments of the literals in C is retrieved from the mode declarations. We now prove that the *armg* construction algorithm terminates and provide a simple example of its application to illustrate the various concepts.

Theorem 4.1. Armg algorithm termination *Assuming the find-first-blocking-literal algorithm terminates, the armg algorithm is guaranteed to terminate for any clause C, background knowledge B and example e.*

Proof. Assume the *armg* construction algorithm does not terminate on at least one input.

However, the algorithm terminates in line 2 if the head of C does not match e. Thus, given the assumption, at least the head of C must match e. We are then forced to execute line 4, setting D to C and entering the 'while' loop of line 5.

For the *armg* algorithm not to terminate, the loop condition, $D \not\models e$, must always hold true. However, assuming the find-first-blocking-literal algorithm of line 6 terminates, at each iteration of the loop, we are removing at least one literal from D. If the loop condition is not verified earlier, D will eventually become bodyless.

Since D was set to C in line 4, and our assumption implies that the head of C must match e, we now have $D \models e$, which contradicts the assumption and completes the proof. □

Example 4.8. Armg construction *Suppose* $C = h(X) \leftarrow b1(X,Y), b2(Y,Z), b3(Z), b4(Y,Z)$, $B = b1(e2,a), b4(a,x)$ *and* $e = h(e2)$. *The mode declarations are:*
$modeh(h(+t)), modeb(b1(+t,-t)), modeb(b2(+t,-t)), modeb(b3(+t)), modeb4(+t,-t)$.

The first blocking literal of C *with respect to example* e *and background knowledge* B *is b2(Y, Z). After removing literal* $b2(Y,Z)$, *literal* $b3(Z)$ *becomes head disconnected and needs to be removed as well. The result of applying armg to clause* C *and example* $h(e2)$ *with respect to background knowledge* B, *is thus clause* $D = h(X) \leftarrow b1(X,Y), b4(Y,Z)$.

First blocking literal

Identifying the first blocking literal is a central part (line 6) of the armg algorithm presented in Algorithm 4.3. We start by defining the first-blocking-literal concept.

Definition 4.9. First blocking literal *Let* B *be background knowledge,* e *an example,* $C = h \leftarrow b_1, \ldots, b_n$ *be a definite clause.* C_i *denotes the subclause of clause* C *which includes all the literals up to* i *(i.e.* $h \leftarrow b_1, \ldots, b_i$*), and* b_i *denotes the body literal at index* i *in* C. b_i *is the first blocking literal of* C, B *with respect to* e *if, and only if,* $C_{i-1}, B \models e$ *and* $C_i, B \not\models e$.

Algorithm 4.4 Find first blocking literal

Require: $C \not\models e$ with respect to B
Input: Clause C, background knowledge B, positive example e
Output: b_i, first blocking literal of C with respect to B and e

```
1: Let lowerbound = 1
2: Let upperbound = length of clause C
3: while lowerbound ≠ upperbound do
4:    Let n = (lowerbound + upperbound)/2
5:    Let C' = clause with first n body literals of C
6:    if C' entails e with respect to B then
7:       lowerbound := n + 1
8:    else
9:       upperbound := n
10:   end if
11: end while
12: return Literal at position lowerbound in clause C
```

In Algorithm 4.4 we present the algorithm to compute the first blocking literal of C that is responsible for C not entailing e with respect to B. The find-first-blocking-literal algorithm is an

adapted binary-search. The algorithm requires at most $\log N$ iterations, where N is the initial length of clause C. In each iteration an entailment test of a subclause of C is performed. Note however that this entailment test, line 6 of Algorithm 4.4, can be quite expensive. For details on how GILPS (and thus ProGolem), deals with entailment tests of long, non-determinate, clauses, see Section 6.2.

4.3.4 Negative-based clause reduction

Negative-based clause reduction is the second refinement operator of ProGolem. The concept of negative-based reduction was initially introduced in Golem [MF92] and later employed in QG/GA [MTN07]. The aim of negative reduction is to generalise a clause by keeping only the literals which prevent negative examples from being proved. Algorithm 4.5 presents the pseudo-code for the negative-based clause reduction of ProGolem.

Algorithm 4.5 Negative-based clause reduction

Input: Clause $C = h \leftarrow b_1, .., b_n$, background knowledge B, set of negative examples E^-
Output: Clause C'. C' is C reduced with respect to E^-

1: Let E_c^- = subset of E^- that clause C covers
2: **while** true **do**
3: Let b_i = first literal of C such that the negative coverage of $h \leftarrow b_1, .., b_i = E_c^-$
4: Let S_{b_i} = literals needed to support b_i (see explanation in the text)
5: Let NS_{b_i} = literals from $b_1, .., b_{i-1} \notin S_{b_i}$
6: Let $C' = h \leftarrow S_{b_i}, b_i, NS_{b_i}$
7: **if** length(C') = length(C) **then**
8: **return** C'
9: **end if**
10: $C \leftarrow C'$
11: **end while**

The rationale behind negative-based reduction is to move to the front of a clause its body literals that are responsible for the clause failing the negative examples. All literals after the failing one can safely be discarded as they do not change the negative coverage of the clause. This process is iterated until there is no reduction in the clause length.

The negative-clause reduction algorithm starts by computing the negative coverage of C, E_c^- (line 1). The reduced clause, C', cannot cover other than E_c^- negative examples. However, by

being shorter, and thus more general, it is likely that the positive coverage of C' is greater than C. At each iteration of the negative-reduction algorithm, we find the first blocking literal, b_i of C such that the negative coverage of $h \leftarrow b_1, .., b_i = E_c^-$ (line 3). Any clause with fewer than i literals would cover more than E_c^- negative examples. This blocking literal is found by a binary-search approach similar to the one of the find-first-blocking-literal algorithm presented in Algorithm 4.4.

After finding b_i we need to compute its support set (line 4). The support set of a literal is the subset of literals b_j, with $1 \leq j < i$, that are needed to connect the head input variables to the input variables of b_i. If the input variables of b_i are all in h then the support set is empty. Note that the support set of a literal may not be unique, see Example 4.10.

Example 4.10. Support set of a literal *Let* $C = h(X) \leftarrow b1(X,Y), b2(X,Y), b3(X,W), b4(Y,Z)$ *and the mode declarations:*
$modeh(h(+t)), modeb(b1(+t,-t)), modeb(b2(+t,-t)), modeb(b3(+t,-t)), modeb(b4(+t,-t))$.
We want to compute the support set for literal $b4(Y,Z)$.

Literal $b4(Y,Z)$ *has one input variable,* Y*, which is not present in the head of the clause. Both* $b1$ *and* $b2$ *introduce variable* Y *(have the variable as their output) and their input variable (*X*) is an input variable from the head of the clause. Therefore, either* $b1$ *or* $b2$ *are possible support sets for* $b4$.

The non-support literals for b_i, NS_{b_i}, are all the literals b_j, with $1 \leq j < i$, that do not belong to the computed support set of b_i (line 5). The reduced clause is now the concatenation of $h \leftarrow S_{b_i}, b_i, NS_{b_i}$ (line 6). Note that the reduced clause retains the non-support literals. The non-support literals are important because the failing literal, b_i, may not be a failing literal in itself, but only in conjunction with other of the literals before it. However, all the literals after b_i in the initial clause can now be safely removed, as their absence does not change the negative coverage of the clause.

If clause C', constructed in line 6, has the same length as in the previous iteration no literal was removed and we can stop the reduction process, returning the reduced clause.

The constraint of requiring the reduced clause to entail no more negative examples than were entailed before reduction may be too strict. For instance, if we allow the reduced clause to cover a few extra negative examples, the reduced clause may become significantly shorter and entail many more positive examples. We would generally prefer this higher compressive clause to a longer, more specific one, that entailed a few less negative examples.

Situations like these are frequent with noisy real-world datasets. ProGolem takes noise into account by allowing the user to choose which scoring function[1] to employ (e.g. compression, precision, accuracy, ...) to determine the (local) optimal b_i of line 3. Note, however, that if one of these custom scoring functions is used the computation of the blocking b_i is now more expensive, as we will need to consider the positive coverage of the clause as well. This customizable negative reduction was not possible in Golem or QG/GA and is a novel contribution of ours.

Time complexity analysis

Theorem 4.2. Negative-based clause reduction worst-case time complexity *The worst-case time complexity for the negative-based clause reduction of ProGolem is* $O((n-k).(c.m.\log n))$*, assuming the cost of a single entailment test is bounded by a constant, c (e.g. maximum number of resolutions allowed); n is the length of the clause to reduce, C; m is the total number of negative examples; and k* $(0 \leq k \leq n)$ *is the number of body literals in the reduced clause, D.*

Proof. In the worst case, only one literal of C is dropped at each iteration, with the blocking literal being the penultimate literal of C. In this worst case there will be $n-k$ reduction iterations. Notice that if the blocking literal is the last literal of C, the reduction algorithm terminates, as the length of the reduced clause will be the same as C. The cost of finding the blocking literal (line 3 of Algorithm 4.5) with respect to all negative examples is $c.m.\log n$ where $c.m$ is the cost of computing the negative coverage of a clause, and $\log n$ is the number of clauses that have to be considered before the blocking literal is found with a binary search. Therefore each iteration has worst-case complexity of $O(c.m.\log n)$. With each iteration costing $c.m.\log n$ and having at most $n-k$ reduction, the worst-case cost is $O((n-k).(c.m.\log n))$. □

[1]The relevant setting is *negative_reduction_measure* of GILPS. See Section 6.4.3 for more information.

Our binary-search approach to identify the blocking literal contrasts with the linear search of Golem and QG/GA. In Golem a linear search is not problematic because the background knowledge is determinate, so there is no backtracking when evaluating whether a clause covers an example. With determinate background knowledge, the complexity of computing coverage of a single clause is $O(n.m)$. The cost of negative reduction in Golem is thus $O((n-k).(n.m))$.

However, when the background knowledge is non-determinate and a linear search is used, as in QG/GA, computing the coverage of a clause costs $O(c.n.m)$. The overall cost for negative reduction is thus $O((n-k).(c.m.n))$ in QG/GA, which compares unfavourably with ProGolem.

It is important to note that, in the calculations above, we are assuming the coverage engine to be Prolog's implementation of SLD-resolution on all ILP systems, with c being the bound on the number of resolutions allowed. Despite not being possible to set apriori a bound on the number of resolutions SLD-resolution will require, the assumption that the number of resolutions is bounded is useful in comparing the time complexity of negative-based clause reduction across these ILP systems.

ProGolem has a further advantage over QG/GA as it can use the sophisticated coverage engines of GILPS, further reducing the computational cost of negative-based clause reduction. These coverage engines are especially useful in non-determinate datasets as in these cases these sophisticated engines generally require several orders of magnitude fewer resolutions than SLD-resolution. See Section 6.2 for a presentation of the coverage engines available in GILPS and a benchmark of their efficiency.

4.3.5 Comparison with other ILP systems

The crucial difference between ProGolem and Progol/Aleph is how the hypothesis space is searched. Aleph performs a top-down search, with literal concatenation being the main refinement (specialization) operator. Literal concatenation specializes a clause in small incremental steps, causing many of the evaluated clauses to be similar.

ProGolem employs a bottom-up search with the *armg* being the main refinement (generaliza-

tion) operator. The *armg* is a powerful generalization operator performing generalization in large leaps. The hypothesis lattice is therefore searched in large steps, visiting many non-similar intermediate clauses.

As a consequence of the top-down approach to searching the hypothesis space, the clauses Aleph generates are smaller and their coverage is relatively cheap to compute. However, computing the coverage of a clause in ProGolem is computationally expensive, requiring specialized coverage engines in the presence of non-determinate background knowledge.

If the anticipated target concept is short and the background knowledge has little non-determinism, Aleph is likely to find the target concept sooner than ProGolem. ProGolem's strength is on domains where the target concept is both large and the background knowledge is non-determinate. In such cases Aleph is likely to be overwhelmed by the size of the hypothesis space and spend all the computational resources in a tiny region of the search space.

The differences between ProGolem and other bottom-up ILP systems lie in the generalization operator used and the control strategy adopted. The closest ILP system to ProGolem is Golem, where the main difference, as mentioned, is the usage of *armg*s in place of determinate *rlgg*s.

More recent bottom-up ILP systems, such as LOGAN-H [AK04], use lgg-like refinement operators but instead of considering all pairs of compatible literals only consider one pair, thus considerably restricting the hypothesis space. LOGAN-H constructs lgg-like clauses by considering only those pairs of literals which guarantee an injective mapping between variables, i.e. it assumes one-to-one object mappings. Other approaches, e.g. [BZB01], use the same idea of simplifying the lgg-like operations by considering only one pair of compatible literals, but this pair is selected arbitrarily.

Another bottom-up ILP system is cBIL [BRS02]. This ILP system, like ProGolem, has the goal of learning long, non-determinate, relational concepts. cBIL even achieves success in more datasets of the Phase Transition framework [GS00] than ProGolem.

Like ProGolem, cBIL requires a specialized θ-subsumption engine to compute the coverage of long, non-determinate, clauses. The motivation for the Django [MS04] θ-subsumption engine

arose from cBIL development. The need to compute coverage of long, non-determinate, clauses also provided the motivation to develop our own θ-subsumption engine, Subsumer [SM10a].

Unlike ProGolem, cBIL has a strong bias on the target hypothesis language. cBIL assumes the following is known and given about the target concept: 1) number of distinct variables, 2) predicate symbols on it, 3) number of times any given literal occurs in the target concept. Furthermore, cBIL cannot deal with noisy datasets. cBIL strategy is to apply constraint-based algorithms to the hypothesis space thus defined.

It is important to note that cBIL learning bias is extremely strong, rendering cBIL virtually inapplicable outside controlled experiments. ProGolem does not require more information about the target concept than Aleph, Progol or Golem.

4.4 Empirical evaluation

4.4.1 Experiment 1 - determinate and non-determinate applications

In this section we empirically compare ProGolem [MSTN09], Golem [MF92] and Aleph [Sri07] on several well-known determinate and non-determinate ILP datasets. The materials to reproduce the experiments in this section, including datasets and programs, are available from http://ilp.doc.ic.ac.uk/ProGolem.

Materials and Methods

Several well-known ILP datasets have been used: Proteins [MKS92], Pyrimidines [KMLS92], DSSTox [RW00], Carcinogenesis [SMS97], Metabolism [CHH$^+$02] and Alzheimers-Amine [KSS95]. The two determinate datasets, Proteins and Pyrimidines, were used with a hold-out test strategy. [2] For the remaining datasets a N-fold cross-validation was performed using the same folds

[2] In a hold-out test strategy a given percentage is used for training and the remaining for test. The original training and test set from [MKS92] (4/5 train) and [KMLS92] (2/3 train) were used. We did not apply N-fold cross-validation to these problems as we were concerned it could favour the classification accuracy due to the training $(N-1)$ folds containing examples too similar to the left-out fold.

as in the original datasets (10 folds for Carcinogenesis, Metabolism and Alzheimers-Amine, 5 folds for DSSTox). Whenever cross-validation was used the accuracy's standard deviation over all the folds is also reported.

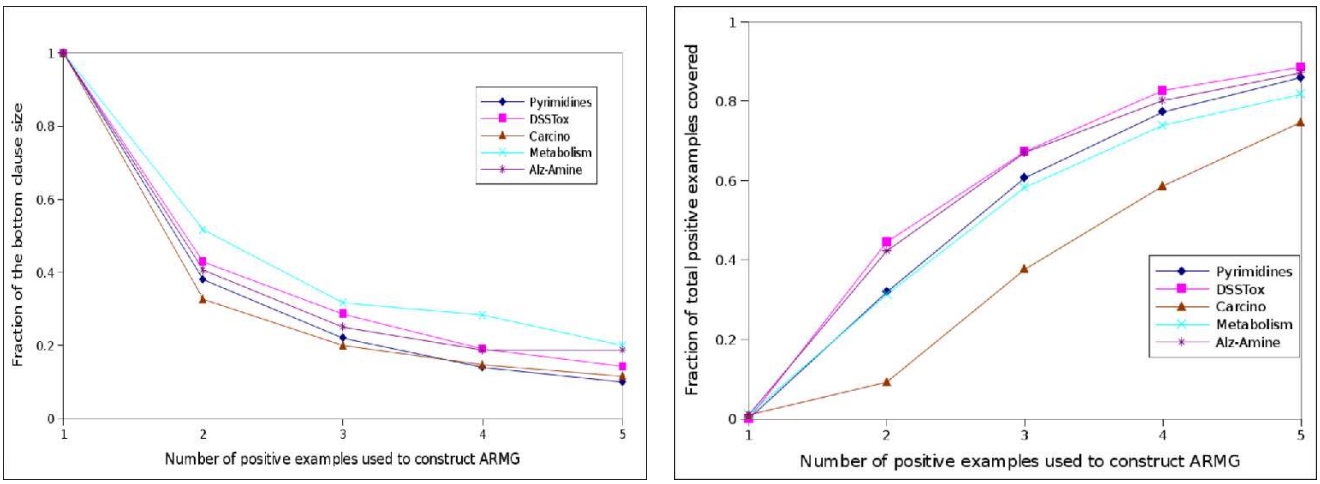
Both Aleph and ProGolem were executed in YAP Prolog 6 with i=2, *maxneg*=10 (except for Carcinogenesis and Proteins where *maxneg*=30) and *evalfn*=compression (except for DSSTox where *evalfn*=coverage). Aleph was executed with *nodes*=1000 and *clauselength*=5 (except for Proteins where *nodes*=10000 and *clauselength*=40). ProGolem was executed with N=2 (beam-width), K=5 (sample size at each iteration) and *negative_reduction_measure* set to precision. ProGolem's coverage testing was Prolog's built-in left-to-right strategy on all these datasets (the same as Aleph). The maximum number of resolutions allowed for ProGolem to prove a clause was set to $10,000$. All experiments were performed on a 2.2 Ghz dual core AMD Opteron processor (275) with 8GB RAM.

	Golem		*ProGolem*		*Aleph*	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
Alz-Amine	N/A	N/A	76.1±4.4	36	76.2±3.8	162
Carcino	N/A	N/A	63.0±7.2	649	59.7±6.3	58
DSSTox	N/A	N/A	68.6±4.5	993	72.6±6.9	239
Metabolism	N/A	N/A	63.9±11.6	691	62.1±6.2	32
Proteins	62.3	3568	62.3	2349	50.5	4502
Pyrimidines	72.1	68	75.3	19	73.7	23

Table 4.1: Predictive accuracies and learning times for *Golem*, *ProGolem* and *Aleph* on different datasets. Golem can only be applied on determinate datasets, i.e. Proteins and Pyrimidines.

Results and discussion

Table 4.1 compares predictive accuracies and average learning times for Golem, ProGolem and Aleph. ProGolem is competitive with Golem on the two determinate datasets. On the Proteins dataset, which requires learning long target concepts, Aleph cannot generate any compressive hypothesis and is slower. This is the type of problem for which a bottom-up ILP system has an advantage over a top-down one. Golem is inapplicable on the remaining non-determinate problems and ProGolem and Aleph have comparable times and accuracies.



(a) (b)

Figure 4.1: (a) *Armg*s length and (b) *Armg*s positive coverage as number of examples used to construct the *Armg*s increases

Figure 4.1 compares the length and positive coverage of *armg*s in ProGolem. In Figure 4.1(a) the *armg* length (as a fraction of the bottom clause size) is plotted against the number of examples used to construct the *armg*. In Figure 4.1(b) the *armg* positive coverage is plotted against the same X axis. When the number of examples is 1, the *armg* (i.e. most-specific clause) coverage is almost invariably the example which has been used to construct the bottom clause and has the maximum length.

As more examples are used to construct the *armg*, the clause length has a rapid decay and the positive coverage a rapid growth. The rapid growth on the positive coverage is due to shorter clauses being more general than longer clauses.

4.4.2 Experiment 2 - complex artificial target concepts

The results of Experiment 1 suggested that for the Proteins dataset, which requires long clauses to be learned, the performance of Aleph is significantly worse than Golem and ProGolem. In this experiment we further examine whether ProGolem has any advantage in situations where the clauses in the target theory are long and highly non-determinate.

Materials and Methods

In this experiment we use a set of eight artificially generated learning problems with varying concept sizes from 6 to 17. These problems are selected from the phase-transition study [BGSS03] and correspond to problems *m6.l12* to *m17.l12*. There are two parameters that characterize each problem in the dataset: the target concept size, m, and the number of distinct constants occurring in an example, L. The number of distinct predicate symbols in a dataset of target concept size m is also m.

The problems *m6.l12* to *m17.l12* are selected from the first row of the *(m,L)* plane, i.e. L=12 so that they only approach the phase-transition region. Each problem has 200 training and 200 test examples and the positive and negative examples are equally balanced in both partitions. We use a hold-out test strategy and compare the performance of ProGolem and Aleph. [BGSS03] consider that the learner has successfully approximated the target concept when the predictive accuracy is $\geq 80\%$.

The phase-transition problem is noise-free and highly non-determinate, having 100 solutions per predicate. The most-specific clause of ProGolem and Aleph defines a bound on the hypothesis space. In order to curb the combinatorial explosion and assess the robustness of the ILP systems to increasing large hypotheses spaces, we varied the recall used in constructing the most-specific clauses. Recall is the maximum number of solutions admitted per unique predicate call of a body literal in the most-specific clause. The recall was set to 1, 2 and 10 for both Aleph and ProGolem. Golem is inapplicable to the phase-transition problem, as the background knowledge for these problems is non-determinate and Golem can handle only determinate datasets.

Aleph and ProGolem were executed with *evalfn*=compression, i=2, *noise*=0. Other Aleph parameters are *clauselength*=20, *search*=heuristic and *nodes*=100,000. ProGolem was executed with N=2 (beam-width), K=10 (sample size). We used three clause-coverage engines in ProGolem: 1) Left-to-right (LFT), which is Prolog's built-in implementation of SLD-resolution and the only coverage engine available in Aleph; 2) smallest-variable domain (SVD) resolution; 3) Subsumer (SUB). See Section 6.2 for details on the coverage engines. In these experiments,

the ProGolem setting *max_resolutions* was set to *inf*. This makes the comparison with SVD and SUB coverages engines fair. It also makes the comparison fair with Aleph as Aleph does not provide a way to bound the number of resolutions a clause may require.

All the experiments were performed, as previously, on a 2.2 Ghz dual core AMD Opteron processor (275) with 8gb RAM.

Results and discussion

Table 4.2 shows the length of the longest most-specific clause for each dataset and for each recall. It is instructive to analyse Table 4.2, as the most-specific clause length is a bound to the size of the hypothesis space.

	Recall			
m	1	2	10	inf
6	12	33	510	601
7	14	39	566	701
8	16	45	661	801
10	20	57	842	1001
11	22	63	931	1101
14	28	81	1212	1401
16	32	93	1498	1601
17	34	99	1608	1701

Table 4.2: Length, for each dataset, of the longest most-specific clause for recalls 1, 2, 10 and infinite

Since the maximum number of solutions per predicate is 100, recall set to *infinite* is the same as 100. It is interesting to note that with recall=10 the most-specific clause is almost as large as with recall=100. The explanation for this is as follows. To add literals to the most-specific clause, the most-specific clause construction algorithm tries all combinations of compatible variables for each predicate symbol. Since for each problem there are m distinct predicate symbols, and each introduces up to *recall* variables, almost all possible combinations of variables are generated even with a recall as low as 10.

Tables 4.3, 4.4 and 4.5 show the predictive accuracies and average learning times, in seconds, for ProGolem and Aleph using recalls 1, 2 and 10 respectively.

r=1	*ProGolem*			*Aleph*	
m	Acc. (%)	T(s) SVD	T(s) LFT	Acc. (%)	T(s) LFT
6	99.5	3	3	99.5	0
7	100	5	5	100	2739
8	100	6	5	100	602
10	100	10	9	50.0	18746
11	98.5	12	14	97.0	2816
14	82.0	21	13	71.5	14607
16	85.5	22	24	50.0	12679
17	61.5	51	54	50.0	16083

Table 4.3: Predictive accuracies and learning times for *ProGolem* and *Aleph* on a set of learning problems with varying concept sizes from 6 to 17, recall=1

The surprising result of Table 4.3 is that for at least 7 of the 8 datasets, the target concept, or good approximations of it, subsumes some most-specific clauses with recall 1. This can be concluded, as ProGolem successfully found 7 predictive theories in this setting.

In contrast, Aleph found only 4 predictive theories and took much longer to do so. There is almost no difference between using SVD or LFT as ProGolem's coverage engine, since the clauses considered are relatively small. Remember that the largest *armg* is the size of the largest most-specific clause and it decays rapidly as more examples are considered.

Aleph took longer in the majority of the datasets because it was forced to exhaust the number of nodes given ($100,000$ clauses per example). This happens because, using a top-down search, the majority of the clauses considered are very similar and have almost identical coverage.

Note that the choice of coverage engine in ProGolem does not impact the learned model. The coverage of a clause is the same, regardless of the coverage engine used. The only difference is the time taken.

Table 4.4 shows the results with recall=2. The average predictive accuracy of ProGolem improved slightly. However, the running times increased significantly and now there is a clear advantage, by a factor of 10, in using SVD rather than LFT as the coverage engine. ProGolem with the LFT coverage engine takes more time than Aleph because the clauses ProGolem considers are longer.

r=2	*ProGolem*			*Aleph*	
m	Acc. (%)	T(s) SVD	T(s) LFT	Acc. (%)	T(s) LFT
6	98.0	8	114	99.5	1
7	99.5	16	413	99.5	254
8	97.5	36	865	100	23
10	97.0	35	1667	50.0	3596
11	99.0	33	146	50.0	3708
14	93.5	46	167	96.0	365
16	76.0	388	9365	50.0	4615
17	71.0	409	5549	50.0	4668

Table 4.4: Predictive accuracies and learning times for *ProGolem* and *Aleph* on a set of learning problems with varying concept sizes from 6 to 17, recall=2

Overall, Aleph's running times decreased and so did its predictive accuracy. Despite the hypothesis space being larger, Aleph spends less time searching hypotheses but still does not manage to find good theories. The explanation for this behaviour is that, when the hypothesis space is larger, the hypotheses vary more and it is easier to overfit the training data. Overfitting is likely to occur with a top-down system because a short hypothesis may easily be marginally compressive without any predictive accuracy. Furthermore, finding a marginally compressive hypothesis reduces the subsequent search effort, as the few positive examples it covers are removed and will not be used to generate new hypotheses.

Note that, with Aleph in particular, there is a negative correlation between the running time and the predictive accuracy. In general, the higher the running time, the lower the predictive accuracy. This is because when a compressive clause is found, all the positive examples the clause covers are removed. The higher the compression of a clause, the more positive examples the clause entails; thus, fewer positive examples are left to derive further hypothesis.

Table 4.5 shows the results with $recall = 10$. Here, only the Subsumer engine of ProGolem was used, as all the other coverage engines were prohibitively slow. For all the datasets except $m = 10$, ProGolem found theories with predictive accuracy $\geq 90\%$. For dataset $m = 10$, ProGolem got lost with very similar *armg*s.

This happened because when the *armg*s are very large and specific, i.e. early in their construction, their coverage may be identical. This prevents ProGolem from assessing which *armg*s are

r=10	*ProGolem*		*Aleph*	
m	Acc. (%)	T(s) SUB	Acc. (%)	T(s) LFT
6	98.0	1025	99.5	0
7	100	2255	97.0	5
8	100	2985	100	9
10	50.0	33519	50.0	3786
11	100	2623	50.0	3969
14	99.5	3713	96.5	1513
16	90.0	17579	50.0	4243
17	97.0	5906	50.0	4537

Table 4.5: Predictive accuracies and learning times for *ProGolem* and *Aleph* on a set of learning problems with varying concept sizes from 6 to 17, recall=10

most promising, and it is forced to select randomly the seed *armg*s for the next iteration.

Aleph's running times with $recall = 10$ are roughly the same as with $recall = 2$ and so are the predictive accuracies, for the same reasons as before. Aleph managed to find an approximation of the target concept in only 4 of the 8 datasets.

4.5 Conclusions and future work

In this chapter we proposed an asymmetric variant of Plotkin's *rlgg* called *armg*. In comparison to the determinate *rlgg*s used in Golem, *armg*s are capable of representing non-determinate clauses. Compared to Plotkin's *rlgg*, the cardinality of an *armg* is bounded by the length of the initially constructed most-specific clause and decays rapidly as more examples are added. With Plotkin's *rlgg*, the cardinality of the clauses grows exponentially in the number of examples.

An algorithm for constructing *armg*s has been implemented in the ProGolem ILP system. ProGolem combines the most-specific clause construction of Progol with a Golem control strategy which uses *armg* in place of determinate *rlgg*. It is shown that ProGolem has a similar or better predictive accuracy and learning times compared to Golem on two determinate real-world applications where Golem was originally tested.

Moreover, ProGolem was also tested on several non-determinate real-world applications where

Golem is inapplicable. In these applications, ProGolem and Aleph have comparable times and accuracies. ProGolem has also been evaluated on a set of artificially generated learning problems with large concept sizes. The experimental results suggest that ProGolem significantly outperforms Aleph in cases where clauses in the target theory are long and complex.

Our results show that while ProGolem has the advantages of Golem with regard to learning large target concepts, it does not suffer from the determinacy limitation and can be used in problems where Golem is inapplicable. Long, non-determinate, target concepts are more effectively learned by an ILP system such as ProGolem since such clauses are easier to construct using a bottom-up search. Top-down ILP systems such as Aleph tend to limit the maximum complexity of learned clauses, due to a search bias which favours shorter clauses. Learning long target concepts requires an overwhelming amount of search for top-down ILP systems.

As shown in Section 4.4.2, the usage of the sophisticated coverage engines of GILPS is central to ProGolem's ability to compute timely the coverage of long, non-determinate, clauses. Our preliminary study suggests that Prolog's left-to-right heuristic for SLD-resolution is only effective in non or very low determinate datasets. For medium to high non-determinate datasets, a more sophisticated coverage engine such as Subsumer should be used. In Section 6.2 we revisit the issue of the trade-offs between the several coverage engines of GILPS in more detail.

We have also identified that, when the *armg*s are very large, a myopia effect can occur. This myopia occurs when all the clauses, in the set of competing *armg*s at a given early iteration, have the same coverage (e.g. 2 positive examples covered, no negatives). In situations like these the *armg*s selected for the next iteration are chosen at random, which is not intended. This issue should be the object of future research.

It may be worth investigating the properties of generalization operators other than *armg*s and implementing these in ProGolem. Implementing these to-be-researched operators in ProGolem involves minimal effort, as it only requires the definition of the generalization operator. All the rest of ProGolem remains constant.

Chapter 5

Subsumer: A Prolog θ-subsumption engine

In this chapter we introduce Subsumer, a new efficient Prolog θ-subsumption engine. This chapter is an extended version of [SM10a] with more detail on the algorithms and the empirical evaluation.

The chapter is organized as follows. In Section 5.1 we explain the need for subsumption engines, both in general and in particular in ILP. In Section 5.2 the θ-subsumption problem is introduced. In Section 5.3 we present Subsumer's algorithms and overview how these compare with the equivalents in two other state-of-the-art θ-subsumption engines, Django and Resumer2. The empirical results comparing Subsumer with Django and Resumer2 are presented in Section 5.4. Section 5.5 concludes and suggests directions for future research.

5.1 Introduction

θ-subsumption is an important problem in computational logic that is particularly relevant to the Inductive Logic Programming and Theorem Proving communities. Efficient θ-subsumption algorithms also have a far-reaching impact on other areas of Artificial Intelligence, e.g. planning

[Skv06]. In ILP, θ-subsumption is at the core of the coverage test, i.e. it determines which examples a clause entails.

Current state-of-the-art ILP systems are usually developed in Prolog, e.g. Aleph [Sri07] and ProGolem [MSTN09], mainly because many of the algorithms needed for an ILP system are already built-in in a Prolog engine (e.g. unification, backtracking, SLD-resolution).

However, for complex learning problems where predicates are highly non-determinate and the target concept size is large (> 10 literals), Prolog's built-in SLD-resolution is inadequate. In these situations there is a combinatorial explosion of alternative variable bindings and consequently it will often take too long for the Prolog engine to decide whether the given goal succeeds. This is unacceptable for an ILP system, as there will be, typically, 10^3 to 10^6 such complex goals (i.e. putative hypotheses) that need to be evaluated before a final theory is proposed.

ILP algorithms construct hypotheses from a rich hypothesis language and thus have to traverse a large search space. This search requires having to test some metric of the candidate clauses. The metric typically used is compression: positive example coverage minus negative coverage minus clause length. Evaluating compression of a single candidate clause thus requires many subsumption tests, each one of those being potentially very expensive, as they are a query in first-order logic.

This problem is well known to ILP researchers and several techniques have been proposed to alleviate it. These techniques include: combining queries in query packs [BDD+02] to take advantage of the similar structure of the candidate clauses; transforming the clause before execution [CSC+03] so that the transformed clause is more efficient to evaluate; improving the indexing mechanism [CSL07] of the Prolog engine; and stochastic estimation of the clause coverage [SR97].

The subsumption test at the core of the ILP bottleneck received far less attention probably because, for many ILP applications, Prolog's built-in resolution seemed to suffice. However, due to the non-determinism explosion highlighted above, ILP researchers often have to bound

the maximum hypotheses length and recall (i.e. number of solutions per predicate) to relatively small values, which may be preventing better theories from being found.

In the last few years, two efficient subsumption engines, Django [MS04] and Resumer2 [KZ08], were developed. However, these are complex engines, with around $10,000$ lines of source code each, implemented in C and Java respectively, making them unpractical to use within a Prolog-based ILP system. More importantly, both those engines require substantial amounts of memory, sometimes 10 times more memory than the ILP system itself for the same data. This limits their applicability considerably given that, for challenging problems, the ILP system already consumes a sizeable portion of the system's resources.

The motivation for Subsumer was to develop a simple, lightweight, fully general Prolog subsumption engine that could be easily integrated from any Prolog application and, in particular, Prolog implementations of ILP systems. In this work we will show not only how that objective, and in particular low memory footprint, was achieved but also how its runtime performance is superior to Django and competitive with Resumer2.

5.2 The θ-subsumption problem

θ-subsumption is an incomplete approximation to logical implication [Rob65]. While implication is undecidable in general, θ-subsumption is a NP-complete problem [KN86].

Definition 5.1. θ-subsumption *Let C and D be two definite clauses and C' and D' be, respectively, the set of literals in the representation of clauses C and D. We say C θ-subsumes D, denoted by $C \succeq D$, if and only if there exists a substitution θ such that $C'\theta \subseteq D'$.*

Example 5.2. θ-subsumption *Consider clauses C and D:*

$C : h(X_0) \leftarrow l1(X_0, X_1), l1(X_0, X_2), l1(X_0, X_3), l2(X_1, X_2), l2(X_1, X_3)$

$D : h(c_0) \leftarrow l1(c_0, c_1), l1(c_0, c_2), l2(c_1, c_2)$

C θ-subsumes D, with $\theta = \{X_0/c_0, X_1/c_1, X_2/c_2, X_3/c_2\}$.

The θ-subsumption problem is thus, given two clauses, C and D, find a substitution θ such that all literals of C, via the substitution θ, are contained in the set of the literals of D.

5.2.1 θ-subsumption time complexity

Let N and M be the lengths of clauses C and D. A straighforward implementation of θ-subsumption has complexity $O(M^N)$ as we need to map each literal of C (ranging from $1..N$) to a literal in D (ranging from $1..M$). In this approach all variables of a literal in C get bound at each assignment to a literal in D and thus, when $M \approx N$, the subsumption problem may become overconstrained and be easier to test than when M is a fraction of N. This straighforward implementation of θ-subsumption is akin to SLD-resolution [KK71]. Within SLD-resolution all mappings from the literals in C onto the literals in D (for the same predicate symbol) are constructed left-to-right in a depth-first search manner. As Prolog programmers know, the order of the literals in C has a significant impact on SLD-resolution (in)efficiency.

Let V be the set of distinct variables in C and T the set of distinct terms in D. The θ-subsumption problem can be translated to finding a mapping from V to T. This approach has complexity $O(|T|^{|V|})$ which is generally better than $O(M^N)$ as usually the clauses we are interested in have $|T| \ll M$ and $|V| \ll N$. Django, Resumer2 and Subsumer use this approach.

5.3 Subsumer

Subsumer is a publicly available (http://ilp.doc.ic.ac.uk/Subsumer), simple ($\approx$ 1000 lines of Prolog) and fully general θ-subsumption engine that has the expected behaviour of a Prolog implementation, as it does not need to keep state. The Subsumer library exports a predicate, $theta\_subsumes(+subsumer, +subsumee)$, that either fails or succeeds. In case of success the variables in the subsumer clause are bound with the corresponding terms/variables of the subsumee and all possible solutions are returned by backtracking.

5.3.1 Main algorithm

The initial stage in the Subsumer algorithm is the precomputation of datastructures (e.g. lists of predicate symbols, domains of predicates, variables, domains of variables, neighbourhood

of variables) that are useful for efficient subsumption of long clauses. A pseudo-code for Subsumer's *theta_subsumes/2* is presented in Algorithm 5.1.

Algorithm 5.1 Theta subsumes

Input: Two definite clauses C and D. C is the subsumer clause and D the subsumee clause
Output: True if $C \succeq D$ (as side effect binds C variables with D terms), False otherwise

1: $PredSymbols_C$ = unique predicate symbols C
2: $PredSymbols_D$ = unique predicate symbols D
3: **if** $PredSymbols_C \nsubseteq PredSymbols_D$ **then**
4: **return** False
5: **end if**
6: **for each** predicate symbol $PS \in PredSymbols_C$ **do**
7: Dom_{PS} = domain of predicate PS in clause D
8: **end for**
9: $Vars_C$ = unique variables of C
10: $Vars\_Domain$ = Variables in C with respective domains extracted from Dom_{PS}
11: $VarsInfo = \{\}$
12: **for each** variable $V \in Vars_C$ **do**
13: $Literals_V$ = list of all literal indexes of C where V occurs
14: $Neighbours_V$ = list of all variables to which V is a direct neighbour
15: $Domain_V$ = domain of V, computed from $Literals_V$ and Dom_{PS}
16: $VarsInfo = VarsInfo \cup \{\langle Literals_V, Neighbours_V, Domain_V \rangle\}$
17: **end for**
18: **return** solve_component($Vars_C$, $VarsInfo$)

In the trivial case where the subsumer clause contains a predicate symbol that does not exist in the subsumee clause, the subsumption test can fail immediately (line 4). Otherwise the constraint problem of mapping the domains of the variables of the subsumer clause to the terms in the subsumee clauses needs to be solved. We call this constraint-solving problem the *solve_component* algorithm as it solves the subsumption problem by continuously dividing a clause into sets of independent subcomponents. See Section 5.3.4 for more details on clause decomposition. The *solve_component* algorithm is presented in Algorithm 5.2.

The *solve_component* algorithm identifies at each iteration the "most-promising" free (i.e. still unbound) variable, V, to bound from the current component. Note that a component is defined solely by the variables appearing on it. The current heuristic is to pick the variable with the smallest domain. Then the current component is decomposed assuming V has been bound (line 5). The pseudo-code for the *decompose_component* algorithm is detailed in Algorithm 5.4.

In line 6 we iterate over the possible values for V's domain and in line 7 update its neighbour

Algorithm 5.2 Solve component

Input: $VarsInComp$: all variables in the current component, $VarsInfo$: information on $VarsInComp$

Output: All consistent mappings for variables in $VarsInComp$ are returned via backtracking

```
 1: if VarsInComp = {} then
 2:    return True
 3: end if
 4: Let V = most_promising_free_variable(VarsInComp, VarsInfo)
 5: Let SubVarsInComps = decompose_component(VarsInComp, VarsInfo, V)
 6: for each value Val ∈ V's domain do
 7:    Let NVarsInfo = update_vars_domains(V, Val, VarsInfo)
 8:    V := Val
 9:    for each component VComp ∈ SubVarsInComps do
10:       solve_component(VComp, NVarsInfo)
11:    end for
12: end for
13: return False
```

variables domain. This neighbour variable domain update (see Section 5.3.3) is the most expensive part of the algorithm. Each time the domain for a neighbour of V becomes empty we have to backtrack and try a different value for V. This process may be lengthy and may get in deep recursive calls before a backtracking occurs.

Note that the *solve_component* algorithm is natural to parallelize. The natural place is the 'for each' loop at line 9 where we could evaluate several of the clause components in parallel. This type of parallelization has the peculiar property of possibly achieving superlinear (in the number of cores) speed-ups in case the subsumption test fails. This is because if a thread evaluating a component fails, all the other component-evaluation threads running in parallel can stop immediately, as there will be no solution for the whole clause. Unfortunately, however, implementing this parallel algorithm is not easy with current Prolog compilers.[1]

Theorem 5.1. Theta subsumes algorithm termination *The theta subsumes algorithm is guaranteed to terminate for any finite subsumer clause C and finite subsumee clause D.*

Proof. Assume the theta subsumes algorithm does not terminate. However, the algorithm

[1]There are two problems: efficiency and transparency. From our experience, managing the threads explicitly in YAP is inefficient and also obfuscates the structure of the algorithm underneath. The ideal situation would be for Prolog compilers to have native parallel versions of list processing libraries. In our case predicate checklist/2 in library(apply_macros) is the relevant one.

terminates in line 4 of Algorithm 5.1 if the predicate symbols of C are not a subset of the predicate symbols of D. Therefore, the predicate symbols of C must be contained in D and the *solve_component* algorithm is called with the finite set of variables in C and information on their domain and other constraints. For the *solve_component* algorithm not to terminate, either 1) the condition in line 1 of Algorithm 5.2, $VarsInComp = \{\}$, must never hold, or 2) we must never reach line 13 from the initial call to *solve_component*. We now show that either 1) or 2) must be verified. If C subsumes D, there is at least one consistent assignment to each variable V of C. Since at each recursive call there is at least one less variable in $VarsInComp$ (see Algorithm 5.4), eventually the set of variables in $VarsInComp$ becomes empty and 1) is verified. If C does not subsume D then, in particular, there will not be any consistent value for the first "most_promising_variable" of C and 2) will eventually be verified. In either case, the assumption that the subsumption algorithm does not terminate is contradicted. □

Although the algorithm is guaranteed to terminate, it may take a long time for arbitrarily complex clauses C and D. In Section 5.3.5 we perform a time complexity analysis of the algorithm.

5.3.2 Datastructures

The subsumer clause, $C = h \leftarrow b_1, .., .., b_n$, is represented as a list of literals. This clause is preprocessed to gather all the distinct (upon variable renaming) calling patterns for the existing predicate symbols. E.g. $l1(X_0, X_1)$ and $l1(X_1, X_2)$ have the same calling pattern but $l1(X_0, X_1)$ and $l1(X_0, X_0)$ are distinct.

The subsumee clause, $D = e \leftarrow g_1, .., .., g_n$, is given as a list of ground literals representing everything known to be true about e (it is the ground most-specific clause of e with recall set to infinity). This clause is preprocessed to gather the domain of each distinct predicate symbol $p_{s/a}$ (i.e. PredicateName/Arity) in it. The domain of a predicate symbol is the possible list of values, $Val(p_{s/a})$ (a set of tuples), the predicate symbol may take. For instance, we compactly represent clause D, in Example 5.2, as $\{l1/2 : [\langle c_0, c_1\rangle, \langle c_0, c_2\rangle], l2/2 : [\langle c_0, c_2\rangle]\}$, representing,

e.g., that the domain of predicate symbol $l1/2$ is the set of tuples $[\langle c_0, c_1\rangle, \langle c_0, c_2\rangle]$.

The space needed to store clause D's related information is thus: $O(\sum_1^N Val(p_{s/a_i}))$ where N is the number of distinct predicate symbols in D. A necessary condition for subsumption is that all distinct predicate symbols in C also exist in D.

The variables are extracted from C and their initial domain is computed. The initial domain of a variable is the intersection of its individual domains in each of the unique calling patterns where the variable occurs. The domain of a variable in a calling pattern is the set of values the variable may assume in that calling pattern. For instance, in Example 5.2, the initial domain for the variables of clause C, when subsuming clause D, are: $X_0 \in \{c_0\}, X_1 \in \{c_1\}, X_2 \in \{c_2\}, X_3 \in \{c_2\}$.

All direct pairwise variable interactions are also stored. A variable v_1 directly interacts with another variable v_2 if, and only if, they share the same literal in C. For instance, we have the following variable interactions for clause C in Example 5.2: $X_0 : \{X_1, X_2, X_3\}, X_1 : \{X_0, X_2, X_3\}, X_2 : \{X_0, X_1\}, X_3 : \{X_0, X_1\}$.

Let V denote the set of distinct variables in C and $\bar{V}$ denote the average number of variable interactions. Then, this requires $O(|V|.|\bar{V}|)$ space which in the worst case is $O(|V|^2)$.

We also have a datastructure that, for each variable, holds the indexes of the literals where the variable occurs in the clause (clause's head being index 1). For the same clause C from Example 5.2 we then have $X_0 : [1, 2, 3, 4], X_1 : [2, 5, 6], X_2 : [3, 5], X_3 : [4, 6]$.

5.3.3 Variable domain update

In this section we present the algorithm to efficiently update the domain of the neighbouring variables of a variable V when V is about to be assigned a value. The neighbouring variables of V is the set of variables with which V directly interacts (i.e. co-occurs in some literal) in the subsumer clause. The pseudo-code for the variable domain update algorithm is presented in Algorithm 5.3. The update variable domain algorithm is called from line 7 of Algorithm 5.2

and is the most expensive operation in Subsumer.

Algorithm 5.3 Update variable domain

Input: V, variable to ground. Val, value to assign to V. $VarsInfo$: information on variables
Output: Updated domains for the neighbours of V consistent with $V = Val$

1: L_V := literals from the subsumer clause where V occurs
2: N_V := direct neighbour variables of V (i.e. vars which co-occur with V in L_V)
3: Upd_Doms := current domains of N_V from $VarsInfo$
4: **for each** unique call pattern of $V \in L_V$ **do**
5: **for each** neighbour variable W of V in L_V **do**
6: Upd_Dom_W := values for W when $V = Val$ in $L_V \cap Upd\_Dom_W$
7: **end for**
8: **end for**
9: **return** VarsInfo with domains of N_V replaced by Upd_Doms

Lines 1-3 have constant cost, as the information needed has been precomputed and is immediately retrieved from $VarsInfo$. Let CP be the number of unique call patterns of V in L_V, CP_i be the number of times V has value Val in call pattern i, N the number of neighbouring variables of V and D_j the current domain of the j^{th} neighbouring variable of V. Then, the time complexity of the variable domain update algorithm is $O(\sum_{i=1}^{CP}(CP_i.\sum_{j=1}^{N} min(CP_i, D_j)))$.

To illustrate the variable domain update algorithm over a unique pattern call, suppose the updating variable is V_C, $Val = c_2$ and the pattern call is $l = l0(V_A, V_B, V_C)$ which has as possible values $\{\{a_1, b_2, c_1\}, \{a_1, b_3, c_1\}, \{a_1, b_1, c_2\}\}$. Suppose also the initial domains for the variables are: $V_A \in \{a_1\}, V_B \in \{b_1, b_2, b_3\}, V_C \in \{c_1, c_2\}$.

We now briefly explain the steps needed to propagate the assignment $V_C = c_2$ to the neighbouring variables of V_C, i.e. V_A and V_B.

We need to check in which indexes of l's values $V_C = c_2$ occurs (in this example $index = \{3\}$ only). This information has previously been precomputed and is available in $VarsInfo$. With this data we can now restrict the domain of each neighbouring variable V_i in time proportional to $min(N, K)$. N is the number of occurrences of the particular V_C value (here c_2) in l's list of values and K is the current domain size of the neighbouring variable V_i. For instance, V_A's domain size is 1 and V_B's is 3. This update is done by iterating over the possible indexes for V_C, collecting the distinct values for each of the interacting variables and finally intersecting these values with the current variables domain.

If at any point the running intersection of the new domain of a neighbour variable of V_C becomes empty, we know that the assignment for V_C is inconsistent. We then have to try a different value for V_C.

5.3.4 Clause decomposition

An important factor for the reduced time complexity in Subsumer is clause decomposition. Let $H = h \leftarrow b_1, .., b_i, .., b_N$ and suppose literal b_i succeeds $k_i > 0$ times. The worst-case number of predicate calls is $\prod_{i=1}^{N} k_i$ which, assuming an average branching factor, b, of solutions per literal, leads to a $O(b^N)$ time complexity. For non-determinate clauses (i.e. clauses having literals with $b > 1$) this becomes untractable for relatively small N.

However, when the clause is decomposable in K components of independent literals, the complexity drops from $O(b^N)$ to $\sum_{i=1}^{K} O(b^{N_{g_i}})$, which is $O(b^{maxN_{g_i}})$. N_{g_i} is the number of literals in component i of the clause. The worst-case number of predicate calls clause is no longer exponential in the length of the full clause but exponential in the length of the longest component.

The reasoning can then be applied recursively to the newly found subcomponents. This idea, named once-transformation, was initially presented in [CSC+03]. In Subsumer we implement a variant of the once-transformation with several important differences. In the once-transformation the clause is transformed and independent literals are embedded in once/1 calls. The transformed clause is then called by the Prolog engine. In our approach, the clause is not transformed and the unit of evaluation is a logical variable, not a literal.

Algorithm 5.4 presents the component decomposition algorithm of Subsumer. In Subsumer the component is represented as a list of variables together with information on the variables interactions. The purpose of Algorithm 5.4 is to, given a set of variables $VarsInComp$, information on these variables interactions $VarsInfo$, and a variable $V \in VarsInComp$, compute the subgroups of variables in $VarsInComp$ that will not interact after V gets ground. Note that the set of all the variables in the decomposed components is exactly $VarsInComp \setminus \{V\}$.

After all the components are found these are sorted in increasing order of their number of

Algorithm 5.4 Decompose component

Input: $VarsInComp$: variables in component, $VarsInfo$: information on $VarsInComp$, V: var to ground

Output: $Components$: set of components (each component is a set of variables)

```
1: Let Vars = VarsInComp \ {V}
2: Let Components = {}
3: while Vars ≠ {} do
4:    Let CurComp = {}
5:    Let W = arbitrary variable ∈ Vars
6:    Let ToVisit= {W}
7:    while ToVisit ≠ {} do
8:       CurComp := CurComp ∪ ToVisit
9:       Let ToVisitNext = {}
10:      for each variable v ∈ ToVisit do
11:         Neighbours_v = variables from Vars which share a literal with v (check VarsInfo)
12:         ToVisitNext := ToVisitNext ∪ Neighbours_v
13:      end for
14:      ToVisit := ToVisitNext
15:   end while
16:   Vars := Vars \ CurComp
17:   Components := Components ∪ {CurComp}
18: end while
19: return Components, sorted ascending in the number of variables per component
```

variables before being returned. In this way, smaller components, which in principle are easier to test, are evaluated (in line 10 of Algorithm 5.2) before longer ones. This can speed up the overall subsumption test significantly in case no solution is found for those smaller components.

The important concept in clause decomposition is component independency, which we define and exemplify below.

Definition 5.3. Independent clause components *Two clause components are independent if, and only if, they do not share any (free) variable.*

Note that a clause is only satisfiable (i.e. there is a consistent assignment of values to its variables) if all its components are. Thus if one component has no solutions, then there is no solution for the whole clause. Equally importantly, the different solutions (θ-substitutions) of a component do not impact the solutions of the remaining components, meaning that we can safely skip to the next component as soon as a solution for the current component has been found.

Example 5.4. Clause decomposition

$h(X) \leftarrow a(X,Y), b(X,Z), c(Y,A), d(Y,B), e(Z,C), f(Z,D)$

In Example 5.4 all variables are connected and thus the whole clause is a single component. However, when variable X (the head variable) becomes bound, literals $a(x,Y), c(Y,A), d(Y,B)$ (that is variables $\{A,B,Y\}$) belong to one component and literals $b(x,Z), e(Z,C), f(Z,D)$ (variables $\{C,D,Z\}$) to another. They are independent of each other, as they do not share any common variable. Resumer2 does this level of decomposition only (called the cut-transformation in [CSC+03]). It decomposes a clause when the head variable becomes bound (i.e. at the beginning of the subsumption test). Django does not do any form of clause decomposition.

In Subsumer this decomposition is applied recursively. If variable Y becomes bound next, then component $a(x,y), c(y,A), d(y,B)$ can be further divided into two components $c(y,A)$ and $d(y,B)$. Literal $a(x,y)$ is now fully ground (i.e.has no free variables) and thus no longer belongs to a component.

Also significant is the fact that in Subsumer the independent components are created dynamically rather than statically at the beginning of clause evaluation. Although this has an overhead, it allow us to choose the variable with the smallest domain (or some other promising heuristic) as the splitting variable rather than, as in the once-transformation, an arbitrary variable where no information about its goodness is available. The costs of doing the decomposition dynamically should be more than offset by minimizing early the domain of the variable used.

5.3.5 Time complexity analysis

Let us consider first the particular case where the subsumer clause cannot even be decomposed in independent subcomponents. In this case the subsumption cost is given by:

$$\prod_{i=1}^{|V|} \sum_{j=1}^{D_i} Prop_{ij} \text{ which is } O((Max\_D.Max\_Prop)^{|V|}) \tag{5.1}$$

where $|V|$ is the number of variables in the subsumer clause, D_i is the size of variable i domain,

i.e. the number of possible values for variable i in the subsumee clause, and $Prop_{ij}$ is the cost of propagating the assignment of a particular value j to variable i. See Section 5.3.3 for the complexity analysis of this propagation. Max_D is the size of the longest domain and Max_Prop is the maximum assignment propagation cost.

Notice that, as we ground variables in the subsumer clause, the number of free neighbour variables decreases and so does the assignment propagation cost.

Let us now consider a more general case where, at the initial variable assignment, the clause was divided into SC subcomponents, each with $|V_{SC}|$ variables. Here the subsumption cost is:

$$\sum_{k=1}^{SC} \prod_{i=1}^{|V_{SC}|} \sum_{j=1}^{D_i} Prop_{ij} \text{ which is } O(SC.(Max\_D.Max\_Prop)^{|V_{SC}|}) \tag{5.2}$$

The cost (5.2) is lower than or equal to (5.1) as the cost is now exponential in the number of variables in V_{SC} rather than in V and $V_{SC} \leq V$. Let us now finally consider the most general case where there are $|V|$ variables in the subsumer clause and, at each variable assignment, the clause is divided into SC subcomponents, each of these components with $|V_i|$ variables, $1 \leq i \leq SC$. In this more general case the subsumption cost is given by:

$$Cost(V) = \begin{cases} D_v & |V| = 1 \\ \sum_{i=1}^{SC} \sum_{j=1}^{D_i} Prop_{P_i j}.Cost(V_i) & |V| \geq 2 \end{cases} \tag{5.3}$$

where P_i is the "most-promising" variable in component i (the concept of "most-promising" was introduced by the *solve_component* algorithm of Algorithm 5.2), D_i is the domain of P_i and V_i is the set of variables in component i. $Cost(V)$ is:

$$O(\bar{C}.(Max\_D.Max\_Prop)^{\log|V|}) \tag{5.4}$$

where $\bar{C}$ is the average number of components at each iteration and Max_D is the number of distinct values of the variable with the largest domain. The exponent $\log|V|$ occurs because we are assuming the number of variables is evenly distributed across the C components and

thus the number of variables needed to be evaluated in series is only $\log_C |V|$, which is of the order $\log |V|$.

The cost (5.3) is lower than (5.2) as the lengths of the subcomponents continue to get smaller at each iteration.

Suppose the maximum assignment propagation cost is constant and equal to 1. The relevant variables for (5.1) are thus only the maximum number of values per variable, $|D|$, and the maximum number of variables, $|V|$. Table 5.1 shows the subsumption cost for Formula (5.1) as a function of these two parameters. Subsumption costs can be interpreted as an upper bound on the number of basic operations needed to be performed in the worst-case to complete a subsumption test.

	Maximum values per variable, $\|D\|$				
$\|V\|$	5	10	30	60	100
4	6.10^3	10^4	8.10^5	10^7	10^8
8	4.10^5	10^8	7.10^{11}	2.10^{14}	10^{16}
16	2.10^{11}	10^{16}	4.10^{23}	10^{28}	10^{32}
32	2.10^{12}	10^{32}	2.10^{47}	8.10^{56}	10^{64}
64	5.10^{44}	10^{64}	3.10^{94}	6.10^{113}	10^{128}

Table 5.1: Number of operations as a function of the number of variables of a subsumer clause, $|V|$, and the maximum number of distinct values for the variables domain, $|D|$.

Let us now illustrate the gains of employing single and recursive clause decomposition. To simplify, assume that when clause decomposition is applied, the clause is always decomposed in two independent components each with half the variables of the original clause. Suppose that our original clause, C, had 33 variables and only a single clause decomposition takes place when a first variable, v, of C gets bound. The total subsumption cost is now:

$$Cost(33, |D|) = 2.|D_v|.Cost(16, |D|) \tag{5.5}$$

where number 2 is the number of subcomponents, $|D_v|$ is the number of values of variable v and $Cost(16, |D|)$ is the cost of doing subsumption with 16 variables. For instance, considering the maximum number of values, $|D| = |D_v| = 30$, the subsumption cost by doing one level of clause decomposition decreases from 6.10^{48} to 3.10^{25}.

Now, let us consider the same situation but applying clause decomposition recursively. The total subsumption cost is now:

$$
\begin{aligned}
Cost(33, |D|) &= 2.D.Cost(16, |D|) \\
Cost(16, |D|) &= D.Cost(8, |D|) + D.Cost(7, |D|) \\
Cost(8, |D|) &= D.Cost(4, |D|) + D.Cost(3, |D|) \\
Cost(7, |D|) &= 2.D.Cost(3, |D|) \\
Cost(4, |D|) &= D.Cost(2, |D|) + D.Cost(1, |D|) \\
Cost(3, |D|) &= 2.D.Cost(1, |D|) \\
Cost(2, |D|) &= |D|.Cost(1, |D|) \\
Cost(1, |D|) &= |D|
\end{aligned} \tag{5.6}
$$

Considering $|D|$, the maximum number of values, to be again 30, then $Cost(33, |D|)$ decreases from 3.10^{25} with single clause decomposition to 6.10^{9} with recursive clause decomposition. This is a very significant reduction in the number of operations needed to test subsumption.

5.3.6 Related engines

There are only two other subsumption engines comparable with Subsumer in terms of the complexity of clauses they can handle: Django [MS04] and Resumer2 [KZ08]. Both are complex engines, around $10,000$ lines of source code each, implemented in imperative languages: C and Java respectively.

Common to the three engines are algorithms inspired by the constraint satisfaction framework. All implement some custom form of arc-consistency and constraint propagation. Django requires particularly large quantities of memory because it performs determinate matching, between the literals in the subsumer clause and the literals in the subsumee clause, prior to starting its normal non-determinate matching. The determinate matching is an idea originally presented in [KL94], where signatures (fingerprints) of a literal are computed taking into

account its neighbours (i.e. variables and literals it interacts).

If the same unique fingerprint exists on both clauses for a given pair of literals, these can be safely matched. This has the potential to speed up the subsumption significantly, but is only effective if those unique signatures do indeed exist. These signatures are expensive to compute and, especially, store. Furthermore, to be discriminative they look for second-level neighbours, which has an even higher cost. Resumer2 only tries determinate matching with first-level neighbours, which incurs a considerably lower memory cost. Subsumer does not implement any form of determinate matching.

Because of these signatures both Django and Resumer2 take a significant amount of time to initialize their datastructures, 20 to 30 times more than Subsumer. If we only need to do a single subsumption test, one subsumer clause against one subsumee clause, Subsumer is highly likely to finish before Django or Resumer2 are ready for the test. However, in a realistic application we will have many hypotheses to be tested against many examples and it can pay off to incur those heavy initialization costs.

Django's default variable ordering heuristic is the minimal variable domain divided by the number of variable interactions. In Resumer2 each variable is assigned a weight equal to its number of interactions divided by its domain size, and then variables are selected with probability proportional to their weight. Subsumer uses simply the minimal variable domain. Django also has a meta layer where it tries to adapt its heuristics to the underlying dataset. Resumer2's main novelty on the other side is a randomized restart mechanism inspired by SAT solvers, where if it finds itself stuck for a long time in a subsumption test, it restarts subsumption with a different variable ordering. This is an interesting idea whose impact we will investigate in the next section.

Finally, Subsumer can deal with arbitrary Prolog clauses whereas both Resumer2 and Django can handle only Datalog clauses (i.e. Prolog clauses with no function symbols). In a Prolog implementation, dealing with function symbols comes relatively naturally (although we could have optimized further our algorithm had we opted not to support them) but in C or Java it is a significant extra burden to support.

5.4 Empirical evaluation

In this section we will extensively compare Django, Resumer2 and Subsumer. The goal of these experiments is to compare running times and memory requirements for the three engines on a very challenging benchmark for θ-subsumption engines. In the sections below, when we refer to examples we mean the subsumee clauses, and by hypotheses we mean the subsumer clauses. This analogy is due to the direct translation of clauses' roles to an ILP system.

All the datasets, subsumption engines and scripts to replicate these experiments can be found at http://ilp.doc.ic.ac.uk/Subsumer.

5.4.1 Datasets

The datasets selected to compare the subsumption engines are instances of the Phase Transition (PT) problem [GS00]. This artificial problem was originally developed to be a challenge for relational learners like Inductive Logic Programming systems.

In an ILP system the challenge is to induce a theory (the target concept) that together with the existing background knowledge covers (i.e. entails) the given positive examples and does not cover the negative examples.

The PT problem is a collection of noise-free datasets of varying difficulty, each characterized by two parameters, the size of the target concept, $M \in [5..30]$, and the distinct number of terms, $L \in [12..38]$, present in a subsumee clause. Furthermore each instance is highly non-determinate with 100 solutions per distinct predicate symbol/arity. For each instance there are 200 positive and 200 negative examples evenly divided between train and test sets; and at least one single clause (the target concept) exists that perfectly discriminates between the positive and negative examples (i.e. has 100% accuracy).

The instances belong to three major regions: Yes, No and Phase Transition. In the Yes region the probability that a randomly generated clause will cover an arbitrary example is close to 1,

in the No region it is close to 0 and in the narrow Phase Transition (PT) region the probability drops abruptly from 1 to 0.

We used the same set of instances that were selected in [BGSS03]. These are 43 datasets from the set of 702 possible PT instances ($range(M) * range(L) = 26 * 27 = 702$). The 43 selected instances are good representatives of three regions, 12 instances are from the Yes region, 15 from the No region and 16 from the PT region. Note that in [BGSS03] the purpose of the experiment was to highlight the difficulty a relational learning system has in learning concepts from the PT and No regions, whereas in the present work we are interested in benchmarking subsumption in these datasets.

This dataset is challenging to a relational learner, mainly due to the high non-determinacy and large concept sizes for ILP standards (typically those are between 3 and 6 literals). Evaluating the coverage of a large non-determinate clause is prohibitively expensive with the traditional built-in, left-to-right, depth-first search implemention of SLD-resolution in Prolog compilers, which is what most ILP systems use.

5.4.2 Subsumee clauses

Each subsumee clause is a single clause with all facts known to be true about the example it represents. The subsumee clauses were generated by running an ILP system and retrieving the fully ground most-specific clause of the respective example. Note that ILP's most-specific clause is variablized, e.g. $h(X) \leftarrow p(X, Y)$. We had to update the ILP system's most-specific clause generation algorithm so that the actual terms beneath the variables were retrieved, e.g. $h(a) \leftarrow p(a, b)$.

All the 400 examples per dataset instance were used. From the subsumption engine perspective all examples are equal; there is no distinction between positive or negative examples. However, since our hypotheses are biased to cover positive examples, it is a better challenge if subsumee clauses that are less likely to be covered are also included.

Due to the nature of the PT dataset all the examples for a particular instance have the same size

(i.e. number of literals) and the number of distinct predicate symbols is equal to the concept size, M. The number of distinct terms in an example is L. The arity of all predicate symbols is three with the first argument being always the term from the head. All terms in the examples are constants with no function symbols. Figure 5.1 shows an excerpt of a subsumee clause for dataset id=3 (m=18, l=16). The full clause has 801 literals.

$$\begin{aligned} p(d0) \leftarrow & br0(d0, d0\_9, d0\_5), br0(d0, d0\_9, d0\_3), br0(d0, d0\_9, d0\_2), \ldots \\ & br3(d0, d0\_0, d0\_11), br3(d0, d0\_0, d0\_1), br4(d0, d0\_9, d0\_6), \ldots \\ & br7(d0, d0\_0, d0\_3), br7(d0, d0\_0, d0\_15), br7(d0, d0\_0, d0\_13). \end{aligned}$$

Figure 5.1: Excerpt of a subsumee clause for dataset id=3 (m=18, l=16).

5.4.3 Subsumer clauses

The clauses used as subsumers (i.e. hypotheses) were generated using the concept of asymmetric relative minimal generalizations (*armg*) [MSTN09], implemented recently in the bottom-up ILP system ProGolem. Essentially the *armg* algorithm receives a clause C and an example e as input and returns a reduced clause R_c, where all literals from C responsible for not entailing e are pruned. See Section 4.3.3 for full details on the *armg* algorithm.

The hypotheses-generation algorithm employed for this experiment receives a list of (positive) examples and computes the iterative *armg* of all of them. The iterative *armg* of a list of examples is found by computing the (variablized) most-specific clause for the first example and then, using it as the start clause, iteratively applying the *armg* algorithm to the remaining examples.

Naturally, the more examples used to construct an *armg* the smaller (and thus more general) it will be. Furthermore, by construction, an *armg* will at least entail all the examples used in its construction. However note that the end result of an *armg* is still a relatively large and specific clause.

In order to create the *armg*s we used 10 randomly selected lists of 6, 7, 8, 9 and 10 positive

examples, yielding 50 varying length hypotheses (10 hypotheses are *armg*s with 6 positive examples, ..., 10 hypotheses are *armg*s with 10 positives).

We could have used *armg*s of negative examples as well and the results would be identical, but we did not want mixing positive and negative examples in the *armg*. The reason is that we know this dataset is noise free, and since an *armg* covers the examples used in its construction, the *armg* resulting from mixing positive and negative examples would likely be much shorter, thus posing a less interesting challenge. Figure 5.2 shows an excerpt of a subsumer clause, an *armg* of 6 positive examples, for dataset id=3 (m=8, l=16). The full hypothesis has 59 literals. The number of variables in any hypothesis is always equal to the number of distinct terms (L)

$$\begin{aligned} p(A) \leftarrow & br0(A,B,C), br0(A,B,D), br0(A,E,F), br0(A,E,G), br0(A,E,H), \ldots \\ & br1(A,E,O), br1(A,E,N), br1(A,E,L), br1(A,E,J), br2(A,J,K), \ldots \\ & br4(A,H,Q), br4(A,D,F), br4(A,D,C), br5(A,I,N), br6(A,J,Q). \end{aligned}$$

Figure 5.2: Excerpt of a subsumer clause for dataset id=3 (m=18, l=16).

in the examples. This is because the secret target concept, created by the authors of the PT problem, has that property and, due to the way *armg*s are constructed, the property is shared with all the hypotheses. Note that, at each stage during *armg* construction, only the literals that make the clause not entail a positive example are removed.

Note that, since our hypotheses are not random but biased towards covering positive examples, in the Yes, No and Phase transition regions the probabilities for subsumption are not necessarily close to 1, 0 and 0.5. Nevertheless, it is still relevant to divide the dataset into these three regions as the subsumption tests have a region-related difficulty.

5.4.4 Subsumption engines

Subsumer, implemented in YAP Prolog [Cos09], was compared with Django [MS04], implemented in C, and Resumer2 [KZ08], implemented in Java. These are relatively recent subsumption engines; older subsumption engines based on determinate matching [KL94] and maximal

clique searches [SHW96] were not tested because we could no longer find them publicly available. However, in [MS04] they were tested against Django and it clearly outperformed those older engines by several orders of magnitude (speed-ups between 150 times to 1,200 times) in this same dataset.

In that paper Django was tested with randomly generated hypotheses with lengths varying from 10 to 50 literals but always with 10 variables. Our *armg*-based hypotheses range from 29 to 626 literals (see Table 5.2) and the number of variables range from 12 to 31 (L's range), which should pose a much bigger challenge to the subsumption engine.

However, notice that it is not easy to come up with a simple and reliable measure of how difficult a given dataset will be. It depends on many factors: examples length, hypotheses length, average ratio between the latter two, distinct terms in the examples, distinct variables in the hypotheses, distinct predicate symbols, ...

As for Resumer2, we will also test a variant, which we name Resumer1, that has randomized restarts turned off. This experiment is interesting because it directly tests the importance of randomized restarts in this benchmark. Furthermore, by comparing the relative performance of Resumer1 to Resumer2, we can roughly estimate the gains we would obtain if we were to implement randomized restarts in Subsumer.

We compiled Django with gcc 4.1.2, Resumer2 (and Resumer1) with Sun's Java 1.6 and Subsumer with Yap6, all with full optimizations enabled. All experiments were performed in an Athlon Opteron processor 1222 running at 3.0 GHz with 4 GB RAM and a 64 bit build of Linux.

5.4.5 Results and discussion

In Table 5.2 the $|Ex|$ column has the number of literals of each saturated example for a given M and L instance of the PT dataset. Please remember that, due to the way the PT dataset was constructed, all examples for a given instance have the same length. The $|Hyp|$ column has the range of the number of literals of the hypotheses used for subsumption testing.

The CPU time column represents the total time, in seconds, taken by the subsumption engine to do the 50 (hypotheses) * 400 (examples) = 20,000 subsumption tests per particular instance of M and L. Likewise, the RAM column represents the memory, in megabytes, required by the subsumption engine to perform those tests.

A first point that is important to mention is that the four subsumption engines returned the same list of subsumed examples for each instance. This was obviously expected as otherwise there would be at least one faulty implementation. Nevertheless, this is a very strong indication that all algorithms correctly implement θ-subsumption.

Inspecting the results in Table 5.2, the first conclusion is that Django consumes too much memory. It consumes so much memory that it crashed in 29 of the 43 datasets due to the 4Gb memory limit being exceeded. Django could not solve a single dataset from the No region, the most challenging one. This excessive memory consumption is partly due to memory leaks in the Django engine; after each subsumption test, there is a small increment in the memory footprint that is never reclaimed. Also, from a CPU time perspective, Django is clearly behind Resumer1/2 and Subsumer by up to 2 orders of magnitude for the few datasets it managed to finish.

The interesting comparison is between Resumer1, Resumer2 and Subsumer. Although Resumer1 is still faster than Subsumer, the difference is merely, on average, 5%, which can be partially attributed to the underlying programming languages. Also relevant is the fact that standard deviations in Subsumer's running times are about half those of Resumer. More importantly, Subsumer's memory requirements are only a small fraction (1/8 to 1/10) of either Resumer.

Table 5.3 [2] summarizes the results of the four subsumption engines in the three regions of the PT dataset. Table 5.4 presents the same data as Table 5.3 but relative to Subsumer (base 1) rather than in absolute terms.

[2]Note that the PT and Overall columns are misleading in favour of Django as, naturally, we can just take into account the datasets where Django successfully finished. If Django could successfully run in those datasets the CPU times and memory would be even higher.

Region	Dataset					Django		Resumer1		Resumer2		Subsumer	
	Id	M	L	$\|Ex\|$	$\|Hyp\|$	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
	0	5	15	501	47-72	1866	2167	73	432	62	288	228	43
Y	1	6	20	601	38-54	1607	2597	98	318	69	748	207	53
e	2	7	19	701	39-56	2057	3033	137	676	70	419	216	58
s	3	8	16	801	45-79	3040	3461	223	675	79	581	**82**	72
	4	9	15	901	49-84	3594	3801	100	891	85	392	182	61
r	5	10	13	1001	63-144	4175	4071	63	578	64	498	149	77
e	6	10	16	1001	43-75	3677	3067	108	505	90	897	207	68
g	7	11	13	1101	78-152	5800	462	82	475	82	748	188	81
i	8	11	15	1101	49-77	4370	462	85	612	82	802	218	86
o	9	12	13	1201	74-173	6261	890	78	806	78	466	185	105
n	10	13	13	1301	77-166	7110	1323	86	709	89	547	180	105
	11	14	12	1401	118-364	9287	1646	54	615	54	544	140	96
	12	13	31	1310	34-53	N/A	N/A	256	1062	87	846	490	91
N	13	15	29	1513	35-48	N/A	N/A	269	962	145	1241	459	118
o	14	15	35	1513	35-49	N/A	N/A	953	802	156	845	**538**	119
	15	15	38	1515	38-50	N/A	N/A	1164	1364	277	1093	**999**	119
r	16	16	38	1616	37-51	N/A	N/A	2495	1320	333	1009	**670**	96
e	17	18	24	1815	34-52	N/A	N/A	221	904	131	792	416	129
g	18	18	35	1818	37-47	N/A	N/A	750	1415	254	1229	554	129
i	19	19	26	1917	34-53	N/A	N/A	1023	1471	181	1139	**630**	144
o	20	21	18	2111	41-70	N/A	N/A	110	860	91	1117	176	169
n	21	24	20	2417	41-61	N/A	N/A	110	937	86	1154	238	178
	22	25	24	2520	38-52	N/A	N/A	197	1164	142	1098	306	167
	23	27	18	2719	42-69	N/A	N/A	107	1177	94	1276	251	149
	24	29	17	2923	43-73	N/A	N/A	100	1444	100	1322	234	172
	25	29	23	2925	36-61	N/A	N/A	247	1204	101	1488	354	170
	26	29	24	2921	35-54	N/A	N/A	157	1421	132	1388	315	170
	27	6	26	601	36-49	N/A	N/A	861	292	258	936	314	53
P	28	6	28	601	35-45	114946	2596	191	309	139	377	309	53
T	29	7	27	701	35-50	N/A	N/A	335	753	159	457	452	58
	30	7	28	701	33-50	N/A	N/A	179	764	171	1046	409	58
r	31	8	27	801	34-50	42526	3478	213	687	154	768	337	70
e	32	11	22	1109	38-60	N/A	N/A	91	729	74	579	266	77
g	33	11	27	1111	36-53	N/A	N/A	272	871	88	1058	404	78
i	34	13	21	1310	39-57	N/A	N/A	188	928	131	1236	229	92
o	35	13	26	1311	34-52	N/A	N/A	528	1179	94	1220	**377**	91
n	36	14	20	1401	40-64	N/A	N/A	82	636	80	798	229	106
	37	14	24	1414	29-53	N/A	N/A	199	691	132	614	302	106
	38	17	14	1701	59-121	N/A	N/A	79	731	79	789	191	96
	39	17	15	1701	48-98	N/A	N/A	91	938	132	1521	208	148
	40	18	16	1801	49-78	N/A	N/A	118	743	86	898	234	105
	41	19	16	1914	47-89	N/A	N/A	112	779	86	788	201	115
	42	26	12	2601	194-626	N/A	N/A	60	950	60	917	210	173
Avg CPU time, Avg RAM:						15023	2361	301	855	119	883	316	105
Std. Dev. CPU time and RAM:						30576	1225	441	314	62	323	170	40
Max CPU time, Max RAM:						114946	4071	2495	1471	333	1521	999	178

Table 5.2: Performance comparison between Django, Resumer1, Resumer2 and Subsumer on the Phase Transition dataset. CPU times are in seconds, RAM is in Megabytes.

	Region							
	Yes		No		PT		Overall	
Engine	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
Django	4,404	2,248	N/A	N/A	78,736	3,037	15,023	2,361
Resumer1	99	608	544	1,167	225	749	301	855
Resumer2	75	578	154	1,136	120	875	119	883
Subsumer	190	75	442	141	292	92	316	105

Table 5.3: Summary of performance comparison between Django, Resumer1, Resumer2 and Subsumer on each region of the Phase Transition dataset. Average CPU times are in seconds, average RAM is in megabytes.

	Region							
	Yes		No		PT		Overall	
Engine	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
Django	23	30	N/A	N/A	269	33	48	22
Resumer1	0.52	8.1	1.23	8.3	0.77	8.1	0.95	8.1
Resumer2	0.39	7.7	0.35	8.1	0.41	9.5	0.38	8.4
Subsumer	1	1	1	1	1	1	1	1

Table 5.4: Comparison of Django, Resumer1 and Resumer2 relative to Subsumer (base 1)

Resumer2 is clearly best on all regions. It is followed by Resumer1, although Subsumer manages to outperform Resumer1 in the hardest No region by $\approx 23\%$. Notice that randomized restarts are particularly helpful in this region and are solely responsible for the almost 4 times speed-up that Resumer2 has on average over Resumer1. In the easiest Yes region, Subsumer is about 2 times slower than either Resumer and randomized restarts have almost no impact. In the PT region Resumer1 outperforms Subsumer by $\approx 30\%$ and Resumer2 outperforms both by about 2 times showing that, again, randomized restarts are helpful. Randomized restarts are more helpful as the difficulty of the subsumption test increases as, for simple instances, randomized restarts either do not have the time to occur or, if they do, are likely to be overhead.

Overall, Resumer2 clearly outperforms Resumer1 being, on average, 2.5 times faster. Also, the standard deviation for a subsumption test in Resumer2 decreased considerably compared with Resumer1. Notably, this is achieved without increasing the memory footprint. Our experiments are further evidence to the claims of the authors of Resumer2 in [KZ08] that randomized restarts are helpful to reduce expected subsumption time.

It is relevant to point out, however, that the version of Resumer2 (and thus 1) used in our experiments incorporates improvements not available in the original implementation [KZ08].

Namely, the version of Resumer1/2 used in our experiments consumes 55% less memory and is ≈ 20% faster. These improvements were made during Subsumer's development as a result of fruitful discussions with the authors of Resumer1/2. In absolute terms, the memory requirements of Resumer are still very high, though. Notice that the reported results are for a 64 bit operation system. In similar experiments in a 32 bit operating system, the memory required by all subsumption engines is about half.

We did a further experiment to test the extent to which dynamic clause decomposition is important to Subsumer. We disabled dynamic clause decomposition and analysed how Subsumer performed without it (i.e. using only the cut transformation). Although for the Yes and PT regions the dynamic clause evaluation turned out to be mainly overhead (10%-25% slower), for all the problems in the No region it proved essential. Without it, Subsumer would still be working after a few hours. This raises the important question of how, without it, Resumer1 and Resumer2 still manage to complete them with competitive times. This may be due to determinate matching that Subsumer does not implement but Resumer1/2 does. Notice that those problems, especially 14, 15, 16 and 19 (highlighted in bold in Table 5.2), are the ones where Subsumer outperforms Resumer1, which is due to the dynamic clause decomposition.

Naturally the ideas embedded in the engines play the dominating role in the final runtime and memory footprint but compilers also have a relevant role. In a separate experiment we compiled Resumer1 with GNU Java compiler (gcj 4.3.3), also with full optimizations enabled. This version of Resumer1 took 2.5 times longer and required 25% more memory than Resumer1 compiled with Sun's JVM. We then compiled Subsumer with SWI-Prolog (5.6.59) which took 5.5 times longer than with YAP6. These empirical results shed some light on the impact of a compiler's generated code on a program's performance.

5.5 Conclusions and future work

We have presented an efficient Prolog-based subsumption engine that can be of use to several logic-related research communities, e.g. Theorem Proving, Planning and Inductive Logic Pro-

gramming. In particular, and relating to our initial motivation for Subsumer, our experiments have shown that with the usage of specialized subsumption engines the coverage of long, non-determinate, clauses can be computed efficiently. As shown in the previous chapter, efficient coverage computation is essential to enable a bottom-up search strategy on non-determinate datasets on an ILP system such as ProGolem.

In our experiments, we have systematically compared the performance of Subsumer, Resumer1, Resumer2 and Django subsumption engines on a very challenging θ-subsumption benchmark. Subsumer clearly outperformed Django both in time and memory, showing that it is possible to perform efficient θ-subsumption in Prolog. Resumer1 (i.e. Resumer2 with no randomized rapid restarts) is, overall, only 5% faster but requires about 8 times more memory than Subsumer, which may be prohibitive in certain scenarios, e.g. when integrated within an ILP system, where the ILP system itself already requires considerable memory. Furthermore, neither Django nor Resumer can handle function symbols in clauses.

Resumer2 has the same memory requirements as Resumer1 but is about 2.5 times faster. Randomized restarts pay off as shown in [KZ08]. This should be enough incentive to implement a randomized restart strategy in a future version of Subsumer, as it is likely that identical performance gains will be achieved. Dynamic clause decomposition was shown to play a crucial part in the ability for Subsumer to compute subsumption efficiently in the hardest region of the PT dataset, the NO region. It is likely that the performance of Django and Resumer1/2 could be improved by integrating dynamic clause decomposition.

From a strict performance perspective, there would be gains in relaxing Subsumer's auto-imposed constraint of having no state. Namely, hypotheses are often related and have many identical literals, much of the datastructures could be computed once and, at the expense of some memory, running times could be significantly improved.

As for the θ-subsumption problem itself, it is worth verifying whether it could be entirely mapped to a constraint satisfaction problem or a sub-graph isomorphism matching problem. If so, one can then use existing state-of-the-art solvers for those problems and check whether they are any better than custom engines like Resumer2 or Subsumer.

Chapter 6

GILPS: General Inductive Logic Programming System

In this chapter we describe GILPS, the general Inductive Logic Programming System which implements all the contributions of this dissertation.

6.1 Introduction

During the research phase of this dissertation many ideas sprang up to test different aspects of ILP systems. It soon became evident that we would need to develop a generic and modular ILP system rather than a set of distinct systems each with its own identity but having many similarities as well, e.g. hypothesis coverage computation, final theory construction, background knowledge and example handling, and so on.

The main guiding principle in GILPS development has been to ease the addition of new ILP engines in the framework. The features that are specific to a given ILP engine (e.g. TopLog, ProGolem) are isolated in a Prolog module with delimited interfaces to the rest of the GILPS framework. The functionality an existing (or new) ILP engine exports to the GILPS framework is only how, in this engine, an hypothesis is generated from an example. The GILPS framework

does the rest.

The ideas and algorithms specific to the ILP engines TopLog and ProGolem and the subsumption engine Subsumer were described in Chapters 3, 4 and 5, respectively. In this chapter we focus on the infrastructure that GILPS provides to the ILP engines that are implemented in it.

The remainder of this chapter is arranged as follows. In Section 6.2 we introduce and benchmark the several coverage engines implemented in GILPS. Section 6.3 describes the global theory construction feature of GILPS which enables efficient cross-validation and the discovery of theories that do not depend on the ordering of the positive examples. A tutorial on using GILPS from a user perspective is presented in Section 6.4. Section 6.5 concludes and suggests directions for future work.

6.2 Coverage engines

The motivation to develop sophisticated coverage engines in GILPS comes from the development of ProGolem. The intermediate clauses ProGolem considers are often long (i.e. have many literals) and non-determinate rendering SLD-resolution inapplicable.

As early as 1985 it was noted that Prolog's built-in left-to-right, depth-first search heuristic for SLD-resolution (hereafter Left-To-Right) can lead to intolerable inefficiencies [SG85]. These inefficiencies are particularly evident when testing the coverage of long, non-determinate, clauses, such as the ones ProGolem needs to consider due to its bottom-up search strategy.

With the aim of making the coverage computation of long, non-determinate, clauses more efficient, we specifically developed four clause-coverage engines in GILPS.[1].

Despiste being particularly relevant to ProGolem, these clause coverage engines are available to any ILP system in GILPS. However, when used with a top-down ILP system, e.g. TopLog, the

[1]The coverage engine to employ in an ILP problem is defined through GILPS setting *clause_evaluation*. It is also possible to set, for any resolution engine, the maximum number of resolution steps allowed with setting *max_resolutions*.

gains in using these coverage engines are smaller since the clauses a top-down system considers are much shorter than a bottom-up ILP system.

Three of the four clause-coverage engines developed are variants of SLD-resolution and one is a θ-subsumption engine. The three variants of SLD-resolution are: Smallest Predicate Domain (SPD-resolution for simplicity), Smallest Variable Domain (SVD-resolution) and Decomposed Smallest Variable Domain (DSV-resolution).

In [SG85] Smith and Genesereth, motivated by an AI planning application, propose a "cheapest-first" heuristic, making the resolution engine choose, during evaluation, the predicate that has fewest solutions. They also recognize the potential overhead of this re-ordering procedure.

In SPD-resolution the literal which, at each moment in the clause evaluation, has the fewest number of solutions is selected. This heuristic has an overhead compared with the default Left-to-Right but, by failing earlier in the search, handles non-determinate clauses better. The SPD-resolution heuristic is equivalent to the "cheapest-first" heuristic described in [SG85].

SVD-resolution is more sophisticated than SPD-resolution; it computes the domain of each variable (i.e. the set of possible values the variable may take) appearing in the clause to be resolved and, at each moment, binds the variable with the smallest domain to one of its possible values. When a variable is ground the domains of its neighbouring variables need to be updated (as done in Subsumer, see Section 5.3.3). Updating the domains of variables while resolving the clause is an expensive computation and SVD-resolution thus incurs a higher overhead than SPD-resolution.

DSV-resolution is identical to SVD-resolution but decomposes dynamically, if possible, a clause in independent subclauses. These subclauses are smaller and are evaluated recursively with DSV-resolution. If no decomposition is possible, SVD-resolution is applied. The clause decomposition idea is also present in Subsumer. For further details on clause decomposition see Section 5.3.4.

The θ-subsumption engine, Subsumer, improves upon DSV-resolution by compiling all the background knowledge, relative to an example, into a single ground most-specific clause. The

background knowledge respective to each example is thus executed just once and is never called during a subsumption test. This allows the pre-computation of shared data structures that contribute to improved performance. Subsumer is presented in detail in Chapter 5.

Subsumer is a θ-subsumption engine, not a resolution engine, which implies some restrictions and adaptations to its usage as a clause-coverage engine. A subsumption engine takes two clauses as input: the subsumee clause and the subsumer clause. The subsumee clause is the ground most-specific clause of an example, which is compiled only once from the background knowledge. The subsumer clause is a regular hypothesis.

Note that, in contrast to a resolution engine, in a subsumption engine the background knowledge is never called during a subsumption test. This is more efficient but carries some limitations that we discuss below.

6.2.1 Subsumption versus resolution

SLD-resolution is the inference rule in logic programming. It allows the Prolog interpreter to derive all logical consequences of a query. In order to use subsumption to decide if an example e is covered by a clause C, one needs to encode all facts related to that example in a single most-specific clause $\bot_e$ (see below for an example). When used to implement ILP's coverage test, θ-subsumption generates the same solutions as SLD-resolution when the underlying Prolog program (i.e. background knowledge in the ILP setting) is pure Prolog and clause C is not recursive.

If the background knowledge contains non-pure Prolog constructs (e.g. non-constructive comparison operators, cuts, ...) subsumption will only find a subset, usually empty, of the solutions that SLD-resolution finds.

Unfortunately, many real-world ILP datasets express their background knowledge in non-pure Prolog. Often the problem lies with *real* number arithmetic. For instance, consider the program in Figure 6.1.

```
:- modeh(1, active(+molecule)).          active(mol1).  logp(mol1, 3.14).
:- modeb(1, logp(+molecule,-real)).      gteq(X, X):- !.
:- modeb(1, gteq(+real,#real)).          gteq(X, Y):- X>Y.
```

Figure 6.1: Simple ILP program with non-pure background knowledge

The ground most-specific clause for $active(mol1)$ is $active(mol1) \leftarrow logp(mol1, 3.14), gteq(3.14, 3.14)$. Suppose now we have the hypothesis $active(X) \leftarrow logp(X, Y), gteq(Y, 3.05)$. This hypothesis does not subsume the ground most-specific clause as there is no literal $gteq(3.14, 3.05)$ in the ground most-specific clause. However, were we to use SLD-resolution, we would be able to prove the hypothesis with the binding $X = mol1, Y = 3.14$.

The culprit of the problem is that the ground most-specific clause did not capture the full information available from $gteq/2$. There are two problems in capturing this information as a ground most-specific clause. The first problem is that the cut in the first $gteq/2$ definition prevents more solutions from being retrieved, when the second argument of $gteq/2$ is unbound or equal to the first argument. The second problem is that the $>$ comparison operator is not constructive. That is, $>/2$ requires both arguments to be instantiated, not returning in backtracking numbers that verify the condition when one or both of the arguments are unbound.

In situations like these one cannot use a θ-subsumption engine to determine the coverage of a clause but need instead to use a resolution engine.

6.2.2 Benchmark

There are many factors that influence which coverage engine is more suitable to use in a given problem. The main factors are the level of non-determinacy of the background knowledge (i.e. average number of solutions per predicate), length of the hypothesis (i.e. number of literals in its body), number of variables in the hypothesis and average size of a variable domain.

Furthermore, the relationship between these factors and the time taken to compute the coverage of a clause is not simple as the coverage engines employ different strategies and have different tradeoffs. In order to gain some insight into the relative performance of each coverage engine

in real datasets, in this section we will compare the performance of the four coverage engines described previously. All the datasets and scripts to replicate these experiments can be found at http://www.doc.ic.ac.uk/∼jcs06/papers/ilp10.

Materials and Methods

We used a representative subset of the datasets already presented in the previous experiments. Datasets PT.02, PT.15 and PT.31 are problems 02, 15 and 31 of the Phase Transition (PT) framework [BGSS03], representing instances from the Yes, No and PT regions.

Since we want to include Subsumer, a subsumption engine, in our benchmark of coverage engines, only pure Prolog was allowed in the background knowledge. This implied removing or disabling cuts and non-constructive comparison operators in some of the datasets' (e.g. mutagenesis) background knowledge.

For the resolution algorithms, the examples are provided in the background knowledge as usual in ILP. For the subsumption engine, each example is a single (saturated) clause with all facts known to be true about it. Below is a small excerpt of a ground most-specific clause for the mutagenesis dataset. The full clause has 77 literals.

$$\begin{aligned} active(d112) \leftarrow & atm(d112, d112\_9, h, 3, 0.136), atm(d112, d112\_8, h, 3, 0.136), \ldots \\ & atm(d112, d112\_1, c, 22, -0.125), bond(d112, d112\_6, d112\_9, 1), \ldots \\ & bond(d112, d112\_1, d112\_7, 1), bond(d112, d112\_1, d112\_2, 7). \end{aligned}$$

Table 6.1 summarizes important statistics on the datasets used. The columns are: number of examples, average example length, average number of distinct predicate symbols per example, average number of solutions per predicate symbol (assuming its input variables are bound) and average number of distinct terms per example. The latter four columns have the respective standard error associated. The figures in Table 6.1 were generated by computing the full (i.e. infinite recall) ground clauses for each example in each dataset.

Dataset	$\|Ex\|$	Examples Len.	Pred. Symb.	Sols per P. S.	Terms per Ex.
Alz-amine	686	31±0	20±0	1±0	23±0
Carcinogenesis	298	115±4	11±0	5±1	54±1
Mutagenesis	188	83±2	2±0	41±2	48±1
Proteins	2028	287±1	42±0	1±0	36±0
Pyrimidines	2788	50±0	10±0	1±0	22±0
PT.02	400	701±0	7±0	100±0	20±0
PT.15	400	1503±1	15±0	100±0	39±0
PT.31	400	804±0	8±0	100±0	28±0

Table 6.1: Relevant statistics for the examples used per dataset

As can be seen from Table 6.1, from the eight datasets selected, three are highly non-determinate (PT.XX) with exactly 100 solutions per distinct predicate symbol. Datasets Alzheimers-amine, Proteins and Pyrimidines are determinate, with each predicate symbol having at most one solution. Carcinogenesis and Mutagenesis have a medium degree of non-determinism.

ProGolem was used to induce theories for these datasets with all the intermediate hypotheses being collected to be later evaluated by the different coverage engines. When inducing theories, ProGolem's recall was set to 20. This is to limit the complexity of the hypotheses generated.

Since ProGolem is a bottom-up ILP system it is biased towards generating longer clauses. However, because some of these datasets are rather simple and all hypotheses were collected (including ones after negative reduction), many short hypotheses were generated as well. Many of those could have been generated by a classical top-down ILP system like Aleph or Progol. For instance, one of the simpler hypotheses generated for the mutagenesis dataset was $active(A) \leftarrow bond(A, B, C, 1), bond(A, C, D, 2)$.

Table 6.2 summarizes the information on the hypotheses collected. The columns have an identical meaning to Table 6.1 except that column "Literals per Predicate Symbol" is the average number of times a given (distinct) predicate symbol appears on the hypothesis. Note that in a hypothesis the terms are usually variables and not just constants or function symbols.

Dataset	$\|Hyps\|$	Hypotheses Len.	Pred. Symb.	Lits per P. S.	Terms per Hyp.
Alz-amine	328	28±1	18±0	1±0	21±0
Carcinogenesis	161	43±3	6±0	4±0	29±2
Mutagenesis	382	43±1	2±0	21±1	33±0
Proteins	464	75±3	19±0	3±0	21±0
Pyrimidines	1730	42±0	10±0	4±0	32±0
PT.02	444	131±8	5±0	24±0	20±0
PT.15	68	163±32	7±1	25±1	36±1
PT.31	156	119±13	5±0	23±0	27±0

Table 6.2: Relevant statistics for the hypothesis used per dataset

Dataset	Coverage engines									
	Left-to-right		SPD-resolution		SVD-resolution		DSV-resolution		Subsumer	
	Time	T-out	Time	T-out	Time	T-out	Time	T-out	Time	T-out
Alz-amine	0.0	0.00%	0.1	0.00%	0.3	0.00%	0.9	0.00%	0.9	0.00%
Carcinogenesis	3.2	0.45%	0.5	0.01%	0.8	0.00%	2.1	0.00%	1.8	0.01%
Mutagenesis	224	36.9%	19	0.27%	35	0.74%	13	0.16%	9.9	0.03%
Proteins	0.1	0.00%	0.4	0.00%	21	0.00%	19	0.00%	8.8	0.00%
Pyrimidines	0.1	0.00%	0.2	0.00%	0.3	0.00%	1.3	0.00%	1.9	0.00%
PT.02	1987	98.8%	721	25.8%	421	8.53%	165	0.02%	26	0.00%
PT.15	771	97.7%	360	64.3%	327	60.3%	301	0.36%	142	0.37%
PT.31	2289	98.8%	405	52.1%	543	43.3%	314	1.96%	76	0.03%

Table 6.3: Average coverage time, in ms, per dataset per coverage engine, together with percentaged of timed-out coverage tests

Results and discussion

Each coverage engine was used to test the Boolean coverage of a random sample of $20,000$ pairs $\langle hypothesis, example\rangle$ from each dataset. Table 6.3 presents the average times, in milliseconds, per subsumption test. Whenever the subsumption test required more than 5 seconds it was aborted. The "Timed-out" column has the percentage of subsumption tests in these circumstances. To compute the average time all the timed-out tests were ignored.

Table 6.3 shows large differences in the coverage-test costs on the non-determinate datasets. On the determinate datasets, the built-in, left-to-right, depth-first search implementation of SLD-resolution available in the Prolog compiler (YAP) is unrivalled. Nevertheless, the time required by SPD-resolution is competitive. As the degree of non-determinism grows, so does the advantage of Subsumer compared with the other coverage engines. It is important to note that Subsumer rarely timed out. However, the main drawback of Subsumer is its overhead on

the determinate datasets and being unable to handle non-pure background knowledge.

Despite Prolog's built-in SLD-resolution being unrivalled on determinate datasets, it should not be used for mildly non-determinate datasets as its performance quickly degrades, leading to a large fraction of the coverage tests timing out. For medium to highly non-determinate datasets, Subsumer should be used. However, Subsumer is only applicable if the background knowledge is pure Prolog. If that is not the case, then DSV-resolution should be employed as it closely resembles Subsumer.

It could be interesting to study if there are performance gains in using a specific coverage engine per pair $\langle hypothesis, example \rangle$ or whether looking at global properties of the dataset is enough to choose the best engine.

6.3 Global theory construction

There are two modes to running an ILP system in GILPS. The typical one, the only supported by Aleph and Progol, is the cover-set mode of Algorithm A.1 where, after the best hypothesis generated from an example is found, that hypothesis is asserted to the background knowledge and all the examples the hypothesis covers are retracted. The hypotheses generation loop is repeated until there are no more positive examples left. This mode of execution is called the 'incremental' theory construction mode in GILPS. The 'incremental' mode has the advantage of being efficient for learning a theory as, in general, only a small fraction of the positive examples will need to be analysed. It has, however, one significant disadvantage: the induced theory depends on the ordering of the positive examples. It is thus possible, as we show in Section 6.3.1, that the best hypotheses are not generated. This situation may occur if these better hypotheses would be generated by examples that were removed by a previous, less compressive, hypothesis.

GILPS introduces a second mode of theory construction, the 'global' theory construction mode. In the 'global' theory construction mode hypotheses from all the positive examples are initially generated. The final theory is only then constructed and all compressive hypotheses are con-

sidered. Computing all the hypotheses before constructing the induced theory is expensive but may permit better theories to be found. It also has the additional advantage of allowing for efficient cross-validation (see Section 6.3.2).

The algorithm used to construct the final theory from a set of hypothesis is given in Algorithm 6.1. The problem of building a final theory in our setting is similar to the set covering problem, which is known to be NP-complete [CLRS01]. As there is no known efficient algorithm to generate the optimal theory T, an approximation will have to suffice. We opted for a simple greedy algorithm which has several advantages compared with more sophisticated optimization algorithms like simulated annealing or genetic algorithms. A greedy algorithm always returns the same theory T, has no parameters to tune, and is efficient and simple to implement in Prolog.

Algorithm 6.1 Greedy theory construction

Input: H, a set of hypotheses with respective coverage
Output: T, the final theory (a subset of H)

```
 1: Let T = {}
 2: while true do
 3:    Let best_score = score(T)
 4:    Let best_hyp = {}
 5:    for each hypothesis h ∈ H do
 6:       Let cur_theory = T ∪ h
 7:       if score(cur_theory) > best_score then
 8:          best_score := score(cur_theory)
 9:          best_hyp := h
10:       end if
11:    end for
12:    if best_hyp = {} then
13:       return T
14:    else
15:       T := T ∪ best_hyp
16:       H := H \ best_hyp
17:    end if
18: end while
```

The basic idea of our theory construction algorithm is to at each iteration greedily select the hypothesis that further increments the current final theory score. The iterations are repeated until no score improvement occurs.

The data available to building the final theory is a set of hypotheses, H, where each hypothesis

has associated the set of examples from which it was derived, E_h, and the set of examples which it entails, C_h.

The final theory, T, is a subset H' of H that maximizes a given score function. The score function is defined as a function of two metrics of T: 1) the set of examples T covers, C_T, and 2) total number of literals in T. See setting *evalfn* in Section 6.4.3 for a list of possible score functions.

The set of examples covered by a theory, C_T, is the union of the examples covered by its individual hypotheses, so the overlap that may exist in the coverage of the hypotheses is ignored. The total number of literals in a theory is the sum of all literals present in each of its hypotheses. The number of literals present in a hypothesis h is denoted by $|h|$. The ultimate goal of a theory is to achieve the highest accuracy on unseen data, while keeping good comprehensibility. In order to fulfil this goal, the compression-based scoring function for theories is $\sum_{e \in E_{c_T}} weight(e) - \sum_{h \in T} |h|$.

In GILPS examples are allowed to have different weights, the weight associated to an example, $weight(e)$, is user-defined, with positive examples having weight >0 and negative examples weight <0. The rationale behind the compression-based score function is to measure by how many literals a theory compresses a dataset, assuming each example is worth one literal in the theory.

Other authors have also noticed the limitations of the standard cover-set approach typically used by inductive rule learners. In [LKFT04] Lavrac et al., motivated by a subgroup discovery problem (i.e. finding rules that describe clusters of examples), developed a weighted covering algorithm that does not remove examples but instead decreases their weight as these are covered by more and more rules.

A significant problem in using the standard cover-set approach when performing subgroup discovery is that only the first few induced rules may be of interest as subgroup descriptions with sufficient coverage and significance. In the subsequent iterations of the cover-set algorithm, rules are induced from biased example subsets (i.e. subsets containing only examples that are

not covered by previously induced rules), inappropriately biasing the subgroups discovered.

As shown in [LKFT04] through extensive empirical evaluation, a weighted covering approach produces smaller sets of rules and each individual rule has statistically significantly higher coverage when compared to the standard cover-set approach that removes examples as these are covered by a newly induced rule.

However, a weighted covering approach is not directly applicable to our setting as we perform predictive induction, i.e. finding rules that discriminate between positive and negative examples, a form of supervised learning rather than descritive induction (non-supervised learning). Nevertheless, it is worth exploring how updating the weights of the examples during learning could improve the predictive accuracy of the ILP systems in GILPS. For instance, one could, in principle, implement a boosting algorithm like AdaBoost [FS95] within GILPS.

6.3.1 Example order relevance

In ILP systems using a cover-set approach (e.g. Aleph and Progol) the best hypothesis generated from an example is asserted to the background knowledge and the examples this hypothesis covers are retracted. This is efficient for learning a theory as, in general, only a small fraction of the examples will need to be analysed. However, it has a major disadvantage. It is possible that the best hypothesis will not be found. The best hypothesis may not be found with the incremental cover-set approach because the example from which the best hypothesis is derived may have been retracted by a previous, less compressive, hypothesis. Figure 6.2 presents a simple problem, in Progol format, illustrating the example order issue.

Note that there are five positive examples $(e(1), e(2), e(3), e(4), e(5))$, three negative examples $(e(6), e(7), e(8))$, and that covering negatives is allowed (i.e. there is noise). The default clause score function is compression (i.e. positives - negatives - num literals). Aleph and Progol will start by generating all hypotheses derivable from example $e(1)$. The two hypotheses derivable from $e(1)$ are $e(X)$ and $e(X) \leftarrow b(X)$.

Hypothesis $e(X)$ covers all examples, i.e. the five positive examples and the three negative

```
:-modeh(e(+int))? :-modeb(1, b(+int))? :-modeb(1, c(+int))?
:-set(noise, 100)?

e(1). b(1). e(2). b(2). c(2).
e(3). b(3). c(3).
e(4). c(4). e(5). c(5).

:-e(6). b(6). :-e(7). :-e(8).
```

Figure 6.2: Mode declarations and background knowledge illustrating the example order relevance problem in Progol

examples, and has one literal. Thus $e(X)$ has a compression score of $5-3-1=1$. Hypothesis $e(X) \leftarrow b(X)$ covers three positive examples $(e(1), e(2), e(3))$, one negative example $(e(8))$ and has two literals. It therefore has a compression score of $3-1-2=0$.

Since hypothesis $e(X)$ is compressive and has a higher score than $e(X) \leftarrow b(X)$ it will be selected for the final theory. The examples it covers (i.e. all) are retracted and the hypotheses derivable from these examples will not be generated. This is unfortunate because, if the hypotheses derivable from example $e(2)$ were considered, Aleph and Progol would be able to induce $e(X) \leftarrow c(X)$. The hypothesis $e(X) \leftarrow c(X)$ covers four positive examples $(e(2), e(3), e(4), e(5))$ and no negatives, having thus a score of $4-0-1=3$.

Hypothesis $e(X) \leftarrow c(X), d(X)$ is the best possible hypothesis in the hypothesis space defined by the program of Figure 6.2. This hypothesis is not found merely because the order in which the examples are presented. If the first example was $e(2)$ rather than $e(1)$, the hypothesis $e(X) \leftarrow c(X)$ would readily be found.

In GILPS, if the setting *theory_construction* is set to 'global' (the default), the final theory is constructed with hypotheses derived from all examples. This is less efficient but avoids the example order problem and may lead to better theories being found.

6.3.2 Efficient cross-validation

When GILPS is executed with *theory_construction*='global', it is possible to perform efficient cross-validation, irrespective of the particular ILP engine selected. In the 'global' theory con-

struction mode, hypotheses are generated from all the training examples, i.e. irrespective of the fold the examples belong to. Each hypothesis has associated two lists of examples: 1) examples used to generate the hypothesis 2) examples it covers (a superset of the first). Algorithm 6.2 shows the pseudo-code for the built-in efficient cross-validation of GILPS.

Algorithm 6.2 Efficient cross-validation

Input: $Folds$, a set of groups of examples, H, a set of hypotheses with respective coverage
Output: T, a set of theories, one per fold

```
1: Let T = {}
2: for each test fold, tf, ∈ Folds do
3:    Let TrainFolds = Folds \ tf
4:    Let TrainHyps = H \ { hypotheses ∈ H generated exclusively from examples in tf}
5:    for each hypothesis, h, ∈ TrainHyps do
6:       Remove from the coverage of h all the examples belonging to tf
7:    end for
8:    Let T_i = Greedy_theory_construction(TrainHyps) (see Algorithm 6.1)
9:    T := T ∪ T_i
10: end for
```

Output: T

Efficient cross-validation is achieved by running the greedy theory construction algorithm (Algorithm 6.1) on each of the folds, with the corresponding hypotheses adjusted. The hypotheses considered for any given pair ⟨training folds, test fold⟩ are the ones which were generated exclusively by examples in the training folds. The hypotheses which require at least one example from the test fold in order to be generated, cannot be generated from the training folds and are thus ignored. Furthermore, the coverage of the these hypotheses is also adjusted to ignore any example they may cover from the test fold.

This form of cross-validation can efficiently learn per-fold theories, providing statistical metrics (see Section 6.4.5) on the quality of the induced theories without having to run the learning algorithm number of fold times. Aleph and Progol do not have built-in efficient cross-validation but instead rely on external scripts to separate the data into folds and run the ILP system fold times in each of the training folds. Executing the learning algorithm for each of the folds individually is inneficient as much of the work is repeated across folds, e.g. many hypotheses (and respective coverage) are shared between folds.

Nevertheless, GILPS also supports this less efficient form of cross-validation. If GILPS is

executed with *theory_construction*='incremental', the theory is learned from each of the training folds individually as does Aleph and Progol. The only advantage then is that GILPS still automatically computes the statistical metrics of the induced theories in all folds, as it does with efficient cross-validation, not requiring an external script.

6.4 Tutorial

GILPS is free for academic usage and its latest version is available at http://ilp.doc.ic.ac.uk/GILPS. GILPS has been developed in Prolog, and requires YAP to be executed [Cos09]. YAP is a high-performance Prolog compiler that is particularly suitable for ILP implementations. YAP is also freely available for academic usage and its latest version is at http://www.dcc.fc.up.pt/∼vsc/Yap/. GILPS has been tested with YAP 6. Earlier versions of YAP do not have all the necessary features to execute GILPS.

At GILPS' webpage there are a number of datasets to which GILPS has been applied. We suggest that the reader downloads these datasets in order to see how GILPS has been configured to solve these problems. The background knowledge and mode declarations definitions of GILPS are identical to Aleph and Progol. We refer the reader to their manuals ([Sri07, MF01]) for a more thorough description of these common aspects.

6.4.1 Examples definition

The examples definitions in GILPS are slightly different than those in Aleph and Progol. In GILPS the examples are specified by the user in any background knowledge file through the special predicate example/2. The first argument is the example itself (a term) and the second is the example weight (a real number). A positive weight represents a positive example, a negative weight a negative example. E.g. 'example(bind(p2BVW),1).' specifies that the example 'bind(p2BVW)' is a positive example with weight 1. Optionally, the user can use example/3 in order to specify the fold of the example. E.g. 'example(bind(p1T10),-1,3).' specifies that 'bind(p1T10)' is a negative example with weight 1 belonging to fold 3. This is useful when us-

ing the same folds as other researchers have and allow directly comparing the cross-validation results. When the examples are loaded, they are assigned an integer example identifier incrementally, starting from 1.

6.4.2 Commands

Only a small set of commands are needed to use GILPS. GILPS commands are defined in module 'gilps.pl'.

build_theory/0 Starts the learning and possibly cross-validation of the problem read with *read_problem* using the current settings in the ILP engine specified by the *engine* setting.

example/{2,3} Specifies an example. The example definition can take either two or three arguments. See Section 6.4.1.

modeb/{2,3} Specifies a mode body declaration, e.g. modeb(*, has_aminoacid(+pdb, -aminoacid_id, #aminoacid_name)). The first argument is an integer specifying the recall. The symbol '*' means indeterminate recall. The precise value assigned depends on the setting 'star_default_recall'. The third argument is optional and specifies whether the mode body declaration is commutative, e.g. modeb(1, diff_aminoacid(+aminoacid_id, +aminoacid_id),commutative).

modeh/{1,2} Specifies the mode head declaration, e.g. modeh(bind(+pdb)). Optionally, for compatibility with Aleph and Progol, there may be an integer specifying the recall, e.g. modeh(*, bind(+pdb)). However, the recall has no meaning in modeh declaration and is ignored by GILPS.

read_problem/1 Reads a Prolog file, which must define, or load files that define, the background knowledge, mode declarations and examples for the problem at hand. E.g. read_problem('hexose.pl').

sat/1 Display the variablized most-specific clause for the example with id (a positive integer) provided in the first argument. E.g. sat(1).

set/2 The first argument is the setting name, the second is the value. If the second argument is ground, the value of the setting is changed, otherwise it is bound to the variable. E.g. set(i, 4), set(clause_evaluation, X) ? X= 'smallest_predicate_domain'. See 6.4.3 for a list of all possible GILPS settings.

ground_sat/1 Display the ground most-specific clause for the example with id (a positive integer) provided in the first argument. E.g. ground_sat(1).

evaluate_theory/1 Evaluates a previously constructed theory from a file after the background knowledge and examples are read with *read_problem*. E.g. evaluate_theory('theory.pl').

6.4.3 Settings

Below is a description of all the user-definable settings of GILPS. These settings are defined in module 'settings/settings.pl'. Unless otherwise stated these settings are valid on all ILP systems in GILPS.

bottom_early_stop (Default=false) If true and there are output variables in the modeh declaration, stops the construction of the most-specific clause in the earliest layer where all the output variables have already occurred. If false or there are no output variables in the modeh declaration, constructs the most-specific clause (bottom clause) normally up to depth 'i'. This setting is only applicable to ILP engines which use most-specific clauses (i.e. ProGolem and FuncLog).

clause_evaluation (Default=smallest_predicate_domain) Defines the clause_evaluation engine to use when computing the coverage of a clause. This setting is applicable to all ILP systems in GILPS but is particularly important to ProGolem, as ProGolem will generate much larger clauses than a top-down ILP system.

Possible choices for the clause_evaluation engine are:

- 'left_to_right': uses Prolog built-in left-to-right, depth-first search heuristic for SLD-resolution. This is only advisable for very short or determinate clauses.

- 'smallest_predicate_domain': SLD-resolution with a selection heuristic that selects at each moment the literal which has the fewest solutions. This heuristic has an overhead compared with 'left-to-right' but, by failing earlier in the search, handles non-determinate clauses better.
- 'smallest_variable_domain': SLD-resolution with a selection heuristic that selects at each moment the variable which has the fewest possible values. This heuristic has an overhead compared with 'left-to-right' and 'smallest_predicate_domain' but may pay off on non-determinate clauses when the number of distinct variables is much smaller than the number of literals in the clause.
- 'advanced': identical to 'smallest_variable_domain' but decomposes a clause recursively in independent sub-components. This engine has a greater overhead than 'smallest_variable_domain' but may pay off on decomposable non-determinate clauses.
- 'theta_subsumption': performs theta-subsumption, using Subsumer [SM10a], between hypothesis and example. Note that the background knowledge must be pure-Prolog.

A detailed explanation of these coverage engines was presented in Section 6.2. A benchmark on a representative dataset of ILP problems was also presented in Section 6.2.2.

clause_length (Default=4) Defines the maximum number of literals (including the head) of a valid clause. ProGolem ignores this setting but TopLog and FuncLog adhere to it.

cross_validation_folds (Default=1) Number of folds to perform cross-validation. A value of '1' instructs GILPS to use all examples for training. Remember that when an example is specified the user can pre-assign it to a specific fold (see Section 6.4.1). If there is no pre-assigned fold for an example, it will be assigned to a random fold.

cut_transformation (Default=false) If 'true' applies the cut-transformation as specified in [CSC+03]. The cut-transformation can only be applied if 'clause_evaluation=left_to_right'. The purpose of this transformation is to speed-up clause evaluation by transforming the clause before coverage computation. Although still applicable, the cut-transformation

has mostly been superseded by the more sophisticated clause evaluation engines available through the 'clause_evaluation' setting.

engine (Default=progolem) Defines which ILP engine GILPS should use. The possibilities are currently:

- 'toplog': TopLog is a top-down ILP system that uses a logic program instead of mode declarations to define the hypothesis space. See Chapter 3 for further details.
- 'progolem': ProGolem is a bottom-up ILP system using asymmetric relative minimal generalizations as the specialization operator. See Chapter 4 for further details.
- 'funclog': FuncLog is a specialized ILP learner for head output connected predicates (i.e. functions). See [STNM09] for further details.

evalfn (Default=compression) Defines which function to use when scoring a clause. Suppose this clause has NL literals and covers TP true positive examples, FP false positive examples, TN true negative examples and FN false negative examples. The total number of examples, E, is $TP + FP + TN + FN$. The most relevant scoring functions are:

- 'accuracy': $(TP + TN)/E$
- 'compression': $TP - FP - NL$
- 'compression_ratio': $(TP - FP)/NL$
- 'coverage': $TP - FP$
- 'novelty': $TP/N - ((TP + FN) * (TP + FP))/(E * E)$
- 'precision': $TP/(TP + FP)$

While defining the scoring of functions, we took into account the remarks of [LFZ99] and the several functions there mentioned (e.g. novelty). For a full list of available scoring functions see module 'hypotheses/score.pl', where the user can specify his or her own scoring function.

depth (Default=20) Maximum depth for the proof of any clause. This setting is important to ensure the interpreter does not enter an infinite loop when evaluating badly behaved

recursive hypotheses or background knowledge.

example_inflation (Default=1) The weight of each example as specified in the examples definition is multiplied by this factor. Remember that when defining the examples, it is possible to assign a custom weight for each example, therefore allowing some examples to be more important than others (see Section 6.4.1). Also notice that if 'example_inflation' is a negative number, the positive and negative examples swap places. See also *positive_example_inflation* and *negative_example_inflation*.

i (Default=3) Defines the number of layers of new variables when constructing the most-specific clause for an example (see Figure A.2). This setting is ignored by TopLog as it does not need to construct most-specific clauses.

maximum_singletons_in_clause (Default=inf) Maximum number of singleton variables (i.e. variables which just occur once) in a clause. This is a TopLog-specific setting.

maxneg (Default=inf) Maximum (absolute) weight of negative examples that may be covered by a clause to still be considered a valid hypothesis.

max_clauses_per_theory (Default=inf) Maximum number of hypotheses in the final induced theory. The default value of 'inf' allows the addition of whatever number of clauses may be needed, as long as there is an incremental gain in adding these clauses to the current theory. The incremental gain is measured according to the *evalfn* setting and also considers the positive and negative examples the theory covers so far.

max_resolutions (Default=10000) This setting is only applicable when the clause evaluation engine is 'left_to_right'. It defines the maximum number of resolutions allowed before failing a coverage test. The maximum resolutions may be set to 'inf' to ensure proper coverage computation (i.e. infinite resolutions). Keep in mind, though, that if the clause under evaluation is long and non-determinate, it is likely the ILP system may take too long. In this case it is better to use another clause evaluation engine. See setting 'clause_evaluation' for the possible options and trade-offs.

max_uncompressive_examples (Default=20) Maximum number of uncompressive examples (or negative score if other scoring function is being used) allowed before stopping the search and computing the current best theory. This setting is only applicable if 'theory_construction'=incremental.

minacc (Default=0) Minimum percentage accuracy a clause has to have on the training data to be considered a valid hypothesis.

mincov (Default=0) Minimum percentage of the positive examples a clause has to cover to be considered a valid hypothesis.

minimum_singletons_in_clause (Default=0) Minimum number of singleton variables (i.e. variables which just occur once) in a clause. This is a TopLog-specific setting.

minpos (Default=0) Minimum weight of positive examples a clause has to cover to be considered a valid hypothesis.

minprec (Default=0) Minimum percentage of corrected predicted positive examples a clause has to have to be considered a valid hypothesis.

negative_example_inflation (Default=1) Multiplies the weights of all negative examples by this factor. See also *example_inflation* and *positive_example_inflation*.

negative_reduction_measure (Default=precision) This is a ProGolem-specific setting. It defines which metric to maximize when performing negative reduction (see Section 4.3.4). The aim of negative reduction is to generalize a clause by keeping only the literals that block (i.e. prevent) negative examples from being proved. In the ILP systems Golem [MF92] and QG/GA [MTN07] only consistency negative reduction is performed. In 'consistency' mode it is ensured that the reduced clause entails no more negative examples than the original clause. However, this restriction may be too strict, as allowing a small extra negative coverage may significantly improve other clause evaluation metrics (e.g. compression, precision). All the possible scoring functions for *evalfn* are also accepted here. The most relevant possible values are:

- 'auto': maximize the same metric as defined by *evalfn*.
- 'consistency': the negatively reduced clause cannot entail more negative examples than the non-reduced clause.
- 'compression': maximize compression of the reduced clause
- 'precision': maximize coverage of the reduced clause

noise (Default=0.5) Maximum percentage of negative weights a clause may cover to still be considered a valid hypothesis.

nodes (Default=5000) Maximum number of hypotheses that may be derived by a single positive example. This setting is TopLog specific.

output_theory_file (Default='theory.pl') Filename where the induced theory is written. If this file already exists, it will be overwritten.

positive_example_inflation (Default=1) Multiplies the weights of all positive examples by this factor. See also *example_inflation* and *negative_example_inflation*.

print (Default=4) This setting controls the pretty printing of clauses to the stdout. It specifies the number of literals to be displayed per line when showing a clause.

progolem_beam_width (Default=3) Number of clauses (ARMGs) to carry forward to the next iteration of ProGolem's search. See Section 4.3.2 for more information. The bigger the beam-width the more likely it is that a better hypothesis will be found from a given example. However, the search will also take longer. The number of ARMGs that are constructed in each iteration of the search is *beam-width* $\times$ *iteration-sample-size*. See also *progolem_iteration_sample_size*. This setting is ProGolem specific.

progolem_iteration_sample_size (Default=20) Number of examples to randomly select to extend the *beam-width* ARMGs of the current iteration. See Section 4.3.2 for more information. As with the beam-width setting, the bigger the iteration sample size, the more likely it is that a better hypothesis will be found from a given example. However, the search will also take longer. The number of ARMGs that are constructed in each iteration

of the search is *beam-width* × *iteration-sample-size*. See also *progolem_beam_width*. This setting is ProGolem specific.

progolem_mode (Default=single) This setting controls the behaviour of the ProGolem algorithm. Possible values are:

- 'single': this is the default behaviour of ProGolem as explained in Section 4.3.2.
- 'pairs': *progolem_iteration_sample_size* pairs of randomly selected positive examples are constructed and the *progolem_beam_width* best are selected as seeds for the next iteration. This is Golem's mode and is more efficient than 'single' but cannot be used with *theory_construction* = 'global'.
- 'reduce': no ARMGs are generated. The hypothesis generated from an example is the negative reduction of the most-specific clause of that example.

random_seed (Default=7) This is an integer specifying the random seed to be used by the ILP system. If the seed is the same across runs, the same numbers will be generated by the random number generators and results can be reproduced.

randomize_recall (Default=false) This setting impacts the construction of the most-specific clause. For a given mode body declaration, *modeb*, if *randomize_recall*='false' the atoms that will be added to the body of the most-specific clause are the first N matches of the *modeb* declaration against the background knowledge.

If *randomize_recall*='true' the N solutions are randomly selected from the set of all possible matches of the *modeb* declaration. This may be useful when the dataset has a degree of non-determinism higher than the *star_default_recall* and we do not want to introduce a bias to favour the first matches.

In ILP systems which do not have this setting, e.g. Aleph and Progol, it is possible to emulate it by shuffling the background knowledge file.

recall_bound_on_evaluation (Default=inf) This is an experimental setting. When evaluating a literal in a clause, only the first *recall_bound_on_evaluation* solutions for any

literal are considered. The default value of 'inf' considers all the solutions, as is expected from Prolog semantics. A lower value would only consider the first solutions, which could wrongly conclude that a given clause does not cover an example when it does. Note that this setting is unrelated to *star_default_recall*.

remove_negatives (Default=false) This setting is only applicable when *theory_construction* = 'incremental'. If set to true, when asserting a new hypothesis to the theory, and in addition to remove the positive examples covered by this hypothesis clause, the negative examples the hypothesis covers are also removed.

sample (Default=1.0) This is a real number between 0.0 and 1.0 specifying the approximate percentage of the user-supplied examples (both positive and negative) to be used by the ILP system. In order to speed-up the learning in datasets where there are too many examples, it may be useful to use a small fraction of the total examples. Ideally the ILP system should already do some form of sample coverage and that is planned for a future version of GILPS (see Section 8.2.1).

smart_coverage (Default=true) If 'true' and the coverage of a prefix of the clause under evaluation is available, then the coverage of the clause under evaluation is computed on the subset of examples that its longest prefix clause covers. This setting speeds up coverage test considerably as examples not covered by a prefix of a clause are guaranteed not to be covered. This setting increases the memory footprint, however. Currently only TopLog is able to take advantage of *smart_coverage*.

star_default_recall (Default=10) The integer value specifying the recall to which a '*' ought to correspond in a modeb definition. The recall setting is an important setting that is used in the construction of the most-specific clause of an example. A higher recall implies a larger hypothesis space. See Section A.2 for further details on how recall is used in the construction of the most-specific clause.

theory_construction (Default=global) In 'global' construction mode the induced theory is constructed only after all hypotheses, from all the positive examples, have been generated. The other theory construction mode is 'incremental' construction. In this mode the

theory is constructed during the search for hypotheses. Incremental theory construction requires fewer computational resources than global theory construction but is example order dependent and may lead to weaker theories. See Section 6.3 for a thorough discussion on this issue. Aleph [Sri07] and Progol [Mug95b] only support 'incremental' theory construction.

verbose (Default=1) An integer ≥ 0 controlling the verbosity of GILPS. The higher the verbose level, the more information is shown.

6.4.4 Sample problem

This section presents a complete, yet simple, problem to bring together the concepts presented so far. In order to run GILPS there must be at least two Prolog files.

The first file (e.g. 'sample.pl') defines, or loads files that define, the learning problem, i.e. mode declarations, background knowledge and examples. Figure 6.4 shows the Prolog file that defines, in GILPS format, the mode declarations, background knowledges and examples for a sample problem. This is the same problem used to show the example order relevance issue with Aleph and Progol in Section 6.3.1.

```
:-modeh(1, e(+int)). :-modeb(1, b(+int)). :-modeb(1, c(+int)).

b(1). b(2). c(2). b(3). c(3). c(4). c(5). b(6).

example(e(1),1). example(e(2),1). example(e(3),1).
example(e(4),1). example(e(5),1).

example(e(6),-1). example(e(7),-1). example(e(8),-1).
```

Figure 6.3: Sample background knowledge, mode declarations and examples for a problem in GILPS

The second file (e.g. 'run.pl') is a simple Prolog program to load GILPS itself, read the problem file and build the theory.

The settings to change GILPS behaviour can be defined in either file. If they are defined in the execution file they must be executed before the call to *build_theory*. Assuming YAP is available,

```
:- ['GILPS/gilps.pl'].
:- read_problem('sample.pl').
%:- set(engine, toplog).
:- build_theory.
```

Figure 6.4: Script to run GILPS on the sample program of Figure 6.3

the program can then be executed by typing in the command line: *yap -L run.pl*. We remind the reader that a set of more interesting problems is available at GILPS' webpage.

6.4.5 Final theory and statistics

After constructing the theory with the command *build_theory/0*, irrespective of which particular engine was used to construct it, GILPS displays the final theory, its confusion matrix and a set of statistics measuring the quality of the induced theory. Besides being displayed to the standard output, the induced theory is saved to the file specified by the 'output_theory_file' setting. This theory output file is itself a Prolog file and can thus be processed by a Prolog interpreter.

For instance, the final theory generated for the protein-hexose binding problem of Chapter 7 is shown in Figure 6.5.

```
Hypothesis 1/2:
#Literals=6, PosScore=37 (37 new), NegScore=4 (4 new) Prec=90.2% (90.2% new)
bind(A):-
   has_aminoacid(A,B,asp), atom_to_center_dist(B,'CG',5.4,0.5),
   has_aminoacid(A,C,asn), has_aminoacid(A,D,asn), diff_aminoacid(D,C).

Hypothesis 2/2:
#Literals=5, PosScore=30 (22 new), NegScore=1 (1 new) Prec=96.8% (95.7% new)
bind(A):-
   has_aminoacid(A,B,glu), atom_to_atom_dist(B,B,'N','CB',2.4,0.5),
   atom_to_center_dist(B,'CD',5.7,0.5), atom_to_center_dist(B,'CB',7.8,0.5).
```

Figure 6.5: Final theory induced for the protein-hexose binding dataset with the aminoacid representation. See Chapter 7.

Notice that, for each hypothesis, GILPS identifies how many of the positive and negative examples it covers and how many of them are new when taking into account the examples covered by the rules before. For instance, the second hypothesis covers 30 positive examples;

of these 30, 22 are novel and 8 were previously covered by the first hypothesis. The confusion matrix and statistics measures about this theory are shown in Figure 6.6.

```
           |                 Actual                |
 Predicted |           Positive|           Negative|             Totals
-----------|-------------------|-------------------|-------------------
   Positive|             59+/-0|              5+/-0|             64+/-0
-----------|-------------------|-------------------|-------------------
   Negative|             21+/-0|             75+/-0|             96+/-0
-----------|-------------------|-------------------|-------------------
 Totals    |             80+/-0|             80+/-0|            160+/-0

Default accuracy: 50% +/-0.0%
Classifier accuracy: 83.8% +/-0.0%
Recall/Sensitivity: 73.8% +/-0.0% (% of correctly class. positive examples)
Specificity: 93.8% +/-0.0% (% of correctly class. negative examples)
Precision: 92.2% +/-0.0% (% of correctly predicted positive examples)
CorPredNeg: 78.1% +/-0.0% (i.e. % of correctly predicted negative examples)
F1-score: 0.819 +/-0.00 (i.e. 2*Precision*Recall/(Precision+Recall)
Matthews correlation: 0.689 +/-0.00
```

Figure 6.6: Confusion matrix and statistical measures for the performance of the induced theory on the training set

If cross-validation is enabled, a similar theory is constructed N times, once for each fold. An average of this n-fold cross-validation is also shown, together with the respective standard deviations, as in Figure 6.7.

```
           |                 Actual                |
 Predicted |           Positive|           Negative|             Totals
-----------|-------------------|-------------------|-------------------
   Positive|             6+/-1|              0+/-1|              6+/-2
-----------|-------------------|-------------------|-------------------
   Negative|             2+/-1|              8+/-1|             10+/-2
-----------|-------------------|-------------------|-------------------
 Totals    |             8+/-2|              8+/-1|             16+/-3

Default accuracy: 50% +/-0.0%
Classifier accuracy: 83.1% +/-6.6%
Recall/Sensitivity: 72.5% +/-12.9% (% of correctly class. positive examples)
Specificity: 93.8% +/-8.8% (% of correctly class. negative examples)
Precision: 93.3% +/-9.0% (% of correctly predicted positive examples)
CorPredNeg: 78.1% +/-7.8% (i.e. % of correctly predicted negative examples)
F1-score: 0.807 +/-0.09 (i.e. 2*Precision*Recall/(Precision+Recall)
Matthews correlation: 0.687 +/-0.12
```

Figure 6.7: Confusion matrix and statistical measures for the average performance of the induced theory on the test set of all folds

6.5 Conclusions and future work

This chapter has shown the main features of GILPS and presented a tutorial on how to use it. GILPS is aimed both at ILP practicioners and researchers and we hope the first can use GILPS as another predictive learner and the latter can extend GILPS to their particular needs or to answer research questions.

In addition to implementing TopLog and ProGolem, each of these having their own novel contributions, GILPS provides two other important contributions to the state of the art in ILP systems implementations, namely: sophisticated coverage engines and global theory construction with efficient cross-validation.

There are a few directions in which GILPS can be directly improved. A promising direction is the integration of a probabilistic learning scheme, such as naive Bayes, within the ILP engine as has been done in nFOIL [LKR05]. This probabilistic learning within ILP has been shown to lead to statistically significant better theories [LKR05]. Another direction to improve the predictive accuracy of the ILP systems in GILPS is the addition of a boosting algorithm, such as AdaBoost [FS95].

Boosting is a machine learning technique where classifiers (in our case induced theories) are combined to result in a final classifier with a higher predictive power. The principle is that the examples are reweighted after each classifier is built so that misclassified examples have their weight increased. In this way, in subsequent iterations the examples that have been misclassified receive more attention and are more likely to be correctly classified.

An important feature missing in GILPS is sampled clause-coverage computation. Currently the full coverage, i.e. using all training examples, is computed for all the thousands of intermediate clauses generated during the hypotheses search. This is expensive and not strictly needed. A sample of the coverage, with a given confidence interval, for a given confidence level, should be enough to reliably guide the hypotheses search through the refinement lattice.

With a confidence interval of α (e.g. 5%) and a confidence level of β (e.g. 95%) we can be

statistically sure that β of the time the estimated coverage is within α of the true coverage. The need for coverage tests is reduced drastically with sampled clause coverage. Since coverage computation represents the largest portion of the CPU time of an ILP system, the savings gained by using sampled clause coverage translate almost fully to performance gains to the whole ILP system. If α and β are chosen adequately, these savings in coverage tests have almost no impact on the quality of the induced theory.

It may be possible to further reduce the need for coverage tests in situations where we need to compute coverage of competing clauses. This idea came from the multi-armed bandit problem [Whi80]. In the multi-armed bandit problem, we have a set of slot machines, each with an unknown reward probability distribution. We have a fixed set of trials and want to find a good trade-off between exploiting the current best slot machine and exploring new ones. Several good heuristics proven to approximate the optimal solution are known from the statistical literature [Whi80].

The multi-armed bandit problem can be translated to the coverage computation in ILP. The slot machines are competing clauses, each with an unknown coverage. We have a fixed set of coverage tests to find which of these competing clauses should be selected to be further refined.

Another direction to further improve the performance of GILPS is the implementation of randomized restarts during the hypotheses search. This is especially useful when searching a very large hypotheses space where the probability of finding a good compressive clause is small. As shown in [ZSJ06], implementing a randomized restarted search strategy in ILP leads to a reduction in the search cost and more accurate induced theories.

In this dissertation we often compare ILP systems in GILPS (e.g. ProGolem) with Aleph, which sometimes is not easy or even fair because GILPS has many features that are not available to Aleph (e.g. different coverage engines, global theory construction, built-in cross validation). Implementing a Progol-like ILP system in GILPS would be advantageous, as it would allow a better understanding of which features of the ILP system are important to the different aspects of the learning problem.

Chapter 7

Protein-hexose binding application

In this chapter we apply ProGolem to a structural molecular biology problem, namely protein-hexose binding prediction.

The original dataset and motivation for this work are due to Houssam Nassif and David Page at the Department of Computer Science, University of Wisconsin-Madison, USA. In [NAAK$^+$09] these researchers applied Aleph to predict protein-hexose binding sites. Their work was presented at ILP 2009 where we also presented ProGolem [MSTN09]. A fruitful conversation ensued and we agreed to apply ProGolem to this protein-hexose binding problem.

The contributions of the author of this thesis were the re-modelling of the problem, to extend the background knowledge and better bias the hypothesis space; and performing the empirical experiments and statistical tests. The biological interpretation of the rules are due to Michael Sternberg, director of the Centre for Bioinformatics at Imperial College London, and Houssam Nassif.

The chapter is organized as follows. Section 7.1 introduces and explains the motivation for addressing the protein-hexose binding problem. In Section 7.2 the dataset is presented together with a definition of two alternative ILP hypothesis spaces. In Section 7.3 experiments comparing Aleph and ProGolem are performed and the biological interpretation of the rules is provided. Section 7.4 concludes the chapter.

7.1 Introduction and motivation

The elucidation of unifying features of the protein-ligand interactions in systems showing a diversity of interaction modes remains a challenging problem, often requiring extensive human intervention. In this work we present an automated general approach to identify these features using ILP. ILP is applied to study the factors relevant to protein/hexose interactions.

Hexoses are 6-carbon monosaccharides involved in numerous biochemical processes including energy release and carbohydrate synthesis [SBM07]. Several non-homologous families of proteins bind hexoses and there are a diverse set of protein-ligand interactions. Several groups have analysed hexose/protein interactions using non-automated approaches, often employing extensive visualization.

Recently [NAAKK09] used support vector machines, a statistical classifier, to obtain a model to discriminate glucose binders from non-binders. This work was later extended [NAAK$^+$09] using an ILP system, Aleph, to study hexose binding in general. A powerful feature of ILP is that, in addition to prediction, it automatically learns rules which can be readily understood. However, the complexity and size of the hypothesis space often presents computational challenges in search time which limit both the insight and predictive power of the rules found.

This work extends [NAAK$^+$09] in two ways. We have extended the background knowledge of [NAAK$^+$09] to include information on which amino acid an atom belongs to. We have further biased the hypothesis space (see Section 7.2.2) to reduce the search space and increase the likelihood of generating meaningful rules. The second difference is that we employ a newly-developed ILP system, ProGolem [MSTN09], which has been shown to learn better than Aleph in highly non-determinate domains (see Section 4.4.2) such as this hexose-binding application.

The combined usage of an extended background knowledge, a better biased search, and the ILP system ProGolem allowed the discovery of both more accurate and more insightful rules. Another contribution of this work is explaining these rules from a molecular biology perspective. While some of the rules were already known from the literature, others have never been reported but are plausible.

7.2 Problem Representation

7.2.1 Dataset

In this work we used the same dataset as in [NAAK+09]. This dataset consists of 80 protein-hexose binding sites (positive examples) and 80 other, non-hexose-binding, sites.

To retrieve the positive examples, the Protein Data Bank [BWF+00] was mined for proteins crystallized with the most common hexoses: galactose, glucose and mannose [FW04]. Theoretical structures and files older than PDB format 2.1 were ignored. Glycosylated sites and redundant structures (at most 30% overall sequence identity) were also eliminated. The non-redundant positive dataset of 80 protein-hexose binding sites is presented in Table 7.1.

Hexose	PDB ID	Ligand	PDB ID	Ligand	PDB ID	Ligand
Glucose	1BDG	GLC-501	1ISY	GLC-1471	1SZ2	BGC-1001
	1EX1	GLC-617	1J0Y	GLC-1601	1SZ2	BGC-2001
	1GJW	GLC-701	1JG9	GLC-2000	1U2S	GLC-1
	1GWW	GLC-1371	1K1W	GLC-653	1UA4	GLC-1457
	1H5U	GLC-998	1KME	GLC-501	1V2B	AGC-1203
	1HIZ	GLC-1381	1MMU	GLC-1	1WOQ	GLC-290
	1HIZ	GLC-1382	1NF5	GLC-125	1Z8D	GLC-901
	1HKC	GLC-915	1NSZ	GLC-1400	2BQP	GLC-337
	1HSJ	GLC-671	1PWB	GLC-405	2BVW	GLC-602
	1HSJ	GLC-672	1Q33	GLC-400	2BVW	GLC-603
	1I8A	GLC-189	1RYD	GLC-601	2F2E	AGC-401
	1ISY	GLC-1461	1S5M	AGC-1001		
Galactose	1AXZ	GLA-401	1MUQ	GAL-301	1R47	GAL-1101
	1DIW	GAL-1400	1NS0	GAL-1400	1S5D	GAL-704
	1DJR	GAL-1104	1NS2	GAL-1400	1S5E	GAL-751
	1DZQ	GAL-502	1NS8	GAL-1400	1S5F	GAL-104
	1EUU	GAL-2	1NSM	GAL-1400	1SO0	GAL-500
	1ISZ	GAL-461	1NSU	GAL-1400	1TLG	GAL-1
	1ISZ	GAL-471	1NSX	GAL-1400	1UAS	GAL-1501
	1JZ7	GAL-2001	1OKO	GLB-901	1UGW	GAL-200
	1KWK	GAL-701	1OQL	GAL-265	1XC6	GAL-9011
	1L7K	GAL-500	1OQL	GAL-267	1ZHJ	GAL-1
	1LTI	GAL-104	1PIE	GAL-1	2GAL	GAL-998
Mannose	1BQP	MAN-402	1KZB	MAN-1501	1OUR	MAN-301
	1KLF	MAN-1500	1KZC	MAN-1001	1QMO	MAN-302
	1KX1	MAN-20	1KZE	MAN-1001	1U4J	MAN-1008
	1KZA	MAN-1001	1OP3	MAN-503	1U4J	MAN-1009

Table 7.1: Hexose-binding sites (protein and respective hexose ligand)

The Protein Data Bank was mined in a similar way for the 80 negative examples. The negative dataset consists of 22 binding sites that bind hexose-like ligands (e.g. hexose or fructose derivatives, 6-carbon molecules, and molecules similar in shape to hexoses), 27 other-ligand binding sites and 31 non-binding sites. The non-binding sites are surface pockets that look like binding

PDB ID	Cavity Centre	Ligand	PDB ID	Cavity Centre	Ligand
Hexose-like ligands					
1A8U	4320, 4323	BEZ-1	1AI7	6074, 6077	IPH-1
1AWB	4175, 4178	IPD-2	1DBN	pyranose ring	GAL-102
1EOB	3532, 3536	DHB-999	1F9G	5792, 5785, 5786	ASC-950
1G0H	4045, 4048	IPD-292	1JU4	4356, 4359	BEZ-1
1LBX	3941, 3944	IPD-295	1LBY	3944, 3939, 3941	F6P-295
1LIU	15441, 15436, 15438	FBP-580	1MOR	pyranose ring	G6P-609
1NCW	3406, 3409	BEZ-601	1P5D	pyranose ring	G1P-658
1T10	4366, 4361, 4363	F6P-1001	1U0F	pyranose ring	G6P-900
1UKB	2144, 2147	BEZ-1300	1X9I	pyranose ring	G6Q-600
1Y9G	4124, 4116, 4117	FRU-801	2B0C	pyranose ring	G1P-496
2B32	3941, 3944	IPH-401	4PBG	pyranose ring	BGP-469
Other ligands					
11AS	5132	ASN-1	11GS	1672, 1675	MES-3
1A0J	6985	BEN-246	1A42	2054, 2055	BZO-555
1A50	4939, 4940	FIP-270	1A53	2016, 2017	IGP-300
1AA1	4472, 4474	3PG-477	1AJN	6074, 6079	AAN-1
1AJS	3276, 3281	PLA-415	1AL8	2652	FMN-360
1B8A	7224	ATP-500	1BO5	7811	GOL-601
1BOB	2566	ACO-400	1D09	7246	PAL-1311
1EQY	3831	ATP-380	1IOL	2674, 2675	EST-400
1JTV	2136, 2137	TES-500	1KF6	16674, 16675	OAA-702
1RTK	3787, 3784	GBS-300	1TJ4	1947	SUC-1
1TVO	2857	FRZ-1001	1UK6	2142	PPI-1300
1W8N	4573, 4585	DAN-1649	1ZYU	1284, 1286	SKM-401
2D7S	3787	GLU-1008	2GAM	11955	NGA-502
3PCB	3421, 3424	3HB-550			

Table 7.2: Non-hexose-binding sites

sites but are not known to bind any ligand. The negative dataset is presented in Table 7.2.

For each binding-site in the dataset we also have the respective binding-site centre. For hexose-binding proteins, the binding centre is the centroid of the pyranose ring. For non-hexose-binding proteins the binding centre is the centroid of the ligand or the empty pocket.

As in [NAAK$^+$09], we do not represent the full protein as an example, as it would contain too much unhelpful data. Instead, the binding sites are represented as a 10 Ångström [1]-radius sphere centred at the ligand. This means that only the atoms in a neighbourhood of 10 Åof this sphere are present in the background knowledge. Information on the amino acids that these atoms are part of was not present in [NAAK$^+$09] and has been added to the background knowledge in our work. See Section 7.2.3 for an illustration of the background knowledge used.

7.2.2 Hypothesis space

In this section we describe how the hypothesis space was defined from an ILP perspective.

[1] 1 Ångström=10^{-10} meters and is usually denoted by Å.

In ILP we define the format of valid hypotheses through mode declarations. A modeh declaration defines the head of a hypothesis and a modeb declaration defines the literals that may appear in the body of the hypothesis.

We will define two alternative hypothesis spaces. The first, which we will call *atom-only*, is a roughly equivalent simplification of the hypothesis space used in [NAAK+09] containing only background on the atoms and their 3D coordinates. The second, which we will call *amino acid*, contains the relationship between atoms and amino acids in addition to the *atom-only* information. Furthermore, as we will see, the *amino acid* representation also provides a better bias to the hypothesis space.

Atom-only representation

The head of each rule is *bind*(+*site*), where *site* is the PDB ID of a hexose binding site. The body of a rule, in the *atom-only* representation, is defined through the mode declarations of Figure 7.1.

```
modeh(bind(+site)).
modeb(*, has_atom(+site,-atom_id, #name, -coords)).
modeb(1, centre_coords(+site,-coords)).
modeb(1, dist(+coords, +coords, #distance, #tolerance), commutative).
```

Figure 7.1: Mode declarations for the *atom-only* hypothesis space

The *has_atom/4* predicate introduces atoms in the 10 Å neighbourhood of the binding site. The *site* is the unique identifier of the binding site, *atom_id* is the unique identifier of the atom, *name* refers to the name of the atom and *coords* is a triplet *coord(X,Y,Z)* specifying the cartesian coordinates of the atom as provided in the PDB file.

The *centre_coords/2* predicate introduces the coordinates of the binding site centre and the *dist/4* literal specifies that two coordinates are within $distance \pm tolerance$ Å of each other. We set *tolerance* constant to 0.5 Å.

The '*' in the mode body definition for the *has_atom/4* predicate indicates it is non-determinate; that is, it may have multiple solutions for the same input (i.e. PDB ID). The '1' in the mode

body definitions for the predicates *centre_coords/2* and *dist/4* indicate determinacy. Determinate predicates may have at most one solution when their input arguments are ground. The *dist/4* predicate is commutative. The input arguments (i.e. two sets of coordinates) could be interchanged without changing the output (i.e. the distance between coordinates and the tolerance).

The distance literal allows the ILP system to express the 3D conformation of the binding site. However, the inclusion of distances makes the learning time exponential in the number of atoms considered. The number of possible distances grows quadratically with the number of atoms considered. This makes the learning very expensive and we will have to bound the non-determinacy of the *has_atom/4* predicate to a low value (e.g. 5-10) while generating hypotheses. This value is called the recall.

Figure 7.2 shows a hypothesis from the *atom-only* hypothesis space together with the respective English translation.

```
bind(A):-
 has_atom(A,B,'OD1',C),
 centre_coords(A,D), dist(D,C,4.6,0.5),
 has_atom(A,E,'CG',F),
 dist(C,F,1.2,0.5), dist(D,F,5.0,0.5).
```

```
A protein is hexose-binding if:
 it contains an OD1 atom that is 4.6+/-0.5 A
 away from the binding centre and 1.2+/-0.5 A
 away from a CG atom and this CG atom is itself
 5.0+/-0.5 A away from the binding centre.
```

Figure 7.2: Example of a hypothesis and its English translation from the hypothesis space considering *atom-only* mode declarations

Amino acid representation

One of the contributions of this work is the re-modelling of the problem representation and a better bias to the hypothesis space. In the *amino acid* representation, the head of each rule is still *bind*(+*site*) as before; however, there are two important differences in the mode body declarations. Figure 7.3 shows the new mode declarations.

The first important difference is the inclusion of amino acids. For a given *site* the *has_aminoacid/3* literal yields amino acids which have atoms in a neighbourhood of 10 Å of the binding site centre. The *has_aminoacid/3* literal outputs both an *aminoacid_id* and a string representing the three-letter code of the amino acid name.

```
modeh(bind(+site)).
modeb(*, has_aminoacid(+site, -aminoacid_id, #aminoacid_name)).
modeb(1, diff_aminoacid(+aminoacid_id, +aminoacid_id), commutative).
modeb(*, atom_to_centre_dist(+aminoacid_id, #atom_name, #dist, #tolerance)).
modeb(*, atom_to_atom_dist(+aminoacid_id, +aminoacid_id, #atom_name,
                           #atom_name, #dist, #tolerance), commutative).
```

Figure 7.3: Mode declarations for the *amino acid* hypothesis space

The second important difference is a stronger bias to the hypothesis space by imposing that the atoms of the protein cannot appear dangling in a hypothesis. An atom can only appear when relating to the distance to another atom or the binding site centre as ensured, respectively, by the *atom_to_atom_dist/6* and the *atom_to_centre_dist/4* literals. Also, an atom can only be introduced given the amino acid it belongs to, allowing an unambiguous identification of the atom in the PDB file.

We argue that these two differences provide a better bias to the hypothesis space when compared to the *atom-only* representation. For the same recall, the size of the most-specific clauses is larger in the *amino acid* representation than in the *atom-only* representation. This is because the recall in the *amino acid* representation works both at the amino acid level and, for each pair of amino acids, at the atom level. Notice also that, although the *amino acid* representation defines a larger hypothesis space, the hypotheses it defines are less expensive to evaluate as they have a lower degree of non-determinism (i.e. have fewer solutions).

For instance in the *atom-only* representation we need 3 literals to relate the distance between two atoms (i.e. two *has_atom/4* and one *dist/4*). Each of the *has_atom/4* literals may match many of the atoms in the background knowledge, leading to a high degree of non-determinism. The non-determinism is further compounded, as the coordinates of each atom may be used later in other *dist/4* literals. By contrast, in the *amino acid* representation only one literal, *atom_to_atom_dist/6*, is needed.

This representation has a much lower degree of non-determinism for two reasons. Firstly, there are fewer possible matching atoms in the background knowledge, as there is a constraint on the amino acid type of these atoms. Secondly, since the atoms' coordinates are not used as input to a later literal - in fact the atoms' coordinates never appear in a hypothesis - the

non-determinism of the whole clause is further reduced.

The mode declarations for the *amino acid* representation of the hypothesis space also include a *diff_aminoacid/2* literal, which allows expressing that two amino acids are different. This may be relevant in the cases in which there are more than one amino acid of the same type in the neighbourhood of the binding centre and each amino acid needs to be individually identified.

Figure 7.4 shows a hypothesis from the *amino acid* hypothesis space together with the respective English translation. Note that in the English translation of the ILP rules in Figures 7.2 and 7.4 there is the implicit condition that all atoms and amino acids are within a neighbourhood of 10 Å from the binding pocket centre.

```
bind(A):-
 has_aminoacid(A,B,asp),
 atom_to_atom_dist(B,B,'N','OD2',4.6,0.5),
 has_aminoacid(A,C,leu),
 has_aminoacid(A,D,cys),
 atom_to_centre_dist(B,'C',7.6,0.5).
```

```
A protein is hexose-binding if:
 the N and OD2 atoms of an asparagine are
4.6+/-0.5 A away and from each other and
 the C atom of this asparagine is 7.6+/-0.5
 A away from the binding centre. A leucine
 and a cysteine are also present.
```

Figure 7.4: Example of a hypothesis and its English translation from the hypothesis space considering amino acid information

7.2.3 Background knowledge

The background knowledge used in this problem is a good illustration of the expressiveness allowed by specifying a problem within the ILP framework. In this case not only ground facts were used but also more complex rules.

In Figure 7.5 we have a small excerpt of the background knowledge for pdb id 1BDG. The *centre_coords/2* predicate specifies the centroid of the hexose pyranose ring (for the positive examples), or the the centroid of the ligand or the empty pocket (for the negative examples). For instance, pdb id 1BDG is a positive example, thus the centre coordinates in Figure 7.5 refer to the pyranose ring of the hexose (a glucose) pdb 1BDG binds to.

The *has_atom/4* predicate specifies, for each pdb id, a unique atom identifier and the atom name and atom coordinates. The *has_aminoacid/3* predicate specifies, for each pdb id, to which

aminoacid name a given atom identifier belongs.

```
centre_coords(p1BDG, coord(27.012,22.131,64.871)).
has_atom(p1BDG, id_1BDG_a64, 'CD2', coord(22.417,13.3,65.511)).
has_atom(p1BDG, id_1BDG_a64, 'CE2', coord(21.61,14.011,66.377)).
has_atom(p1BDG, id_1BDG_a85, 'C', coord(24.62,25.935,57.444)).
has_atom(p1BDG, id_1BDG_a85, 'O', coord(24.643,24.768,57.845)).
has_atom(p1BDG, id_1BDG_a86, 'N', coord(24.755,26.978,58.261)).
has_atom(p1BDG, id_1BDG_a86, 'CA', coord(24.926,26.781,59.688)).
has_aminoacid(p1BDG, id_1BDG_a64, phe).    has_aminoacid(p1BDG, id_1BDG_a85, leu).
has_aminoacid(p1BDG, id_1BDG_a86, gly).    has_aminoacid(p1BDG, id_1BDG_a87, gly).
```

Figure 7.5: Excerpt of the background knowledge for pdb id 1BDG

Figure 7.6 shows the definition of the *dist/4* predicate. The first definition is of the tolerances allowed for the the distances. We only allow tolerances of 0.5 Å as it is a sensible error margin for the distances between atoms in a protein. This is the same value that was used in [NAAK+09]. Note that for each additional value allowed for the tolerance the hypothesis space grows exponentially.

There are two definitions for the *dist/4* predicate. The first definition is used when the third argument to *dist/4*, the distance between two coordinates, is unbound (i.e. has not been computed yet). In this case the distance is computed and returned together with the tolerance (0.5 Å). Note that the number of decimal places in the distance is truncated to one. This truncation is done to generalize the actual distance. The actual distance with many decimal places is specific to an example and is thus meaningless and should not appear in a hypothesis. The second definition for the *dist/4* predicate is used when the distance (and tolerance) between two coordinates is given. In this case the distance is computed and the predicate only succeeds if the computed distance is in fact within the given distance ± tolerance.

It is important to provide both definitions in the background knowledge. The first definition is used during hypothesis generation. When hypothesis are being generated only the input arguments (specified with a '+' in the mode declarations) are ground. Constant arguments (specified with a '#' in the mode declarations), such as distance and tolerance, are unbound. The second definition is used to compute the coverage of a hypothesis. When we are testing whether a hypothesis entails an example the constant arguments are ground and the coordinates are the ones from the example being tested for entailment. Both definitions of *dist/4* need to

compute the Euclidean distance between a pair of 3D points. The predicate defining this is the *euc_dist/3*.

```
tolerance(0.5). %tolerance

%dist/4 definition used for hypothesis generation
dist(Coord1, Coord2, Dist, Tolerance):-
  Coord1\=Coord2,
  var(Dist), !,
  euc_dist(Coord1, Coord2, Dist1),
  Dist is truncate(Dist1*10)/10,%to get just 1 decimal place
  tolerance(Tolerance).
%dist/4 definition used for coverage testing when Dist is ground
dist(Coord1, Coord2, Dist, Tolerance):-
  number(Dist), number(Tolerance),
  euc_dist(Coord1, Coord2, Dist1),
  abs(Dist1 - Dist) =< Tolerance.

euc_dist(coord(X1,Y1,Z1),coord(X2,Y2,Z2), Dist):-
  Dist is sqrt((X1 - X2)^2 + (Y1 - Y2)^2 + (Z1 - Z2)^2).
```

Figure 7.6: Definition of the dist/4 predicate

Figure 7.7 shows the definition of the predicates *atom_to_centre_dist/4* and *atom_to_centre_dist/6* used in the mode declarations of the *amino acid* representation. These predicates are rules constructed in terms of the ground facts *has_aminoacid/3*, *has_atom/4*, *centre_coords/4* and the *dist/4* predicate. As explained previously, the usage of the predicates *atom_to_centre_dist/4* and *atom_to_centre_dist/6* bias more strongly the hypothesis space.

```
atom_to_centre_dist(Aminoacid_id, Atom_name, Dist, Tolerance):-
  has_aminoacid(PDB, Aminoacid_id, _),
  has_atom(PDB, Aminoacid_id, Atom_name, Coords1),
  centre_coords(PDB, Coords2),
  dist(Coords1, Coords2, Dist, Tolerance).

atom_to_atom_dist(Amin_id1, Amin_id2, A_name1, A_name2, Dist, Tolerance):-
  has_aminoacid(PDB, Amin_id1, _),
  has_atom(PDB, Amin_id1, A_name1, Coords1),
  has_aminoacid(PDB, Amin_id2, _),
  has_atom(PDB, Amin_id2, A_name2, Coords2),
  dist(Coords1, Coords2, Dist, Tolerance).

diff_aminoacid(A, B):- A\=B.
```

Figure 7.7: Clauses in background knowledge

7.3 Experiments

7.3.1 Methods

We have used two ILP systems, Aleph [Sri07] and ProGolem [MSTN09], with both *atom-only* and *amino acid* representations.

The systems were configured with as close as possible settings to ensure a fair test, although some settings are inevitably system specific. Both Aleph and ProGolem were executed under the same system with the same Prolog interpreter, YAP 6.0.6 [Cos09]. The common settings are: recall = 7, maxneg = 5 (i.e. maximum number of negatives a hypothesis may cover) and evaluation function = compression. The compression evaluation scores a clause according to: *positive examples covered - negative examples covered - clause length.*

ProGolem-specific settings are: beam-width = 2, iteration sample size = 5, negative reduction measure = precision, theory construction = global and clause evaluation = smallest variable domain resolution. The beam-width and iteration sample size settings control the amount of resources given to the beam-search over the hypothesis space. See Section 4.3.2 for a detailed explanation of these settings.

Global theory construction ensures the theory is only constructed after all hypotheses have been generated. Aleph always constructs the theory incrementally, which may lead to poorer choices of hypotheses. See Section 6.3 for further details. The negative reduction measure specifies which metric to maximize when performing negative reduction. See Section 4.3.4 for further details.

The clause evaluation setting specifies the coverage engine. Due to the size and non-determinism of the clauses, Prolog's built-in SLD-resolution is not adequate, often timing out. See Section 6.2 for further details on the coverage engines available to ProGolem.

Aleph-specific settings are: nodes = 5000 (maximum hypotheses to derive from an example) and clause length = 5 for the *atom-only* representation and clause length = 6 for the *amino*

acid representation. Clause length is the maximum number of literals in a hypothesis. The clause length for Aleph was set after noticing the size of the hypothesis ProGolem generated. Notice that if the same clause length was used in both representations, the predictive accuracies of Aleph would be lower. In ProGolem the user does not need to specify the maximum clause length.

All materials (i.e. dataset, ILP systems and scripts) to reproduce these experiments are available at http://ilp.doc.ic.ac.uk/Hexose.

7.3.2 Results

It is important to note that the main aim of this work is to discover rules describing the bio- and stereo-chemistry of protein-hexose binding. Although there is empirical evidence suggesting that many hexose dockings are not accompanied by substantial protein conformational changes [SB04], we do not aim to predict the binding sites of new hexoses, as we would not know in advance the coordinates of the binding site. Nevertheless, predictive accuracies are used as a measure to demonstrate the quality of the rules.

We use a 10-fold cross-validation to train and test our approach. We divide the data set into 10 stratified folds, thus preserving the proportions of the original set labels and sub-groups.

Table 7.3 shows the 10-fold cross-validation predictive accuracies of Aleph and ProGolem with the *atom-only* and *amino acid* representations. The folds used are the same as those in [NAAK$^+$09], so the fold-by-fold results are directly comparable. We have also included the backward-selection k-Nearest Neighbor (BS kNN). BS kNN is the best reported classifier in [NAAK$^+$09].

Table 7.4 summarizes the predictive accuracies of ProGolem, Aleph and BS kNN. Notice that BS kNN is a statistical classifier requiring a constant-length feature vector as input. This requires a different problem representation than the one used with ILP. Essentially the input to the BS kNN consists of atom counts for 3 concentric layers composing the binding site and extending 5 Å, 3 Å and 2 Å respectively. Full details are available in Section 6.1 of [NAAK$^+$09].

Fold	1	2	3	4	5	6	7	8	9	10	Avg	Std Dev.
Aleph 1	50.0	68.8	62.5	50.0	75.0	68.8	75.0	93.8	68.8	56.3	66.9	13.2
ProGolem 1	75.0	81.3	68.8	56.3	81.3	87.5	81.3	81.3	75.0	56.3	74.4	10.8
Aleph 2	56.3	68.8	68.8	68.8	56.3	81.3	75.0	75.0	75.0	87.5	71.3	9.8
ProGolem 2	75.0	81.3	93.8	75.0	81.3	87.5	81.2	93.8	81.3	81.3	83.2	6.6
BS kNN	75.0	81.3	81.3	62.5	68.8	81.3	75.0	81.3	68.8	75.0	75.0	6.6

Table 7.3: 10-folds cross-validation predictive accuracies for Aleph and ProGolem. 1) *atom-only* representation, 2) *amino acid* representation.

	ProGolem	Aleph	BS kNN
Atom-only	$74.4\% \pm 10.8\%$	$66.9\% \pm 13.2\%$	$75.0\% \pm 6.6\%$
Aminoacid	$83.2\% \pm 6.6\%$	$71.3\% \pm 9.8\%$	

Table 7.4: Mean predictive accuracy and standard deviation for ProGolem, Aleph and BS kNN

Table 7.3 has enough data to answer two relevant questions: 1) is there any gain in using the *amino acid* representation over that of *atom-only*? 2) are ProGolem and Aleph accuracies equivalent for the same representation? In order to answer these questions we performed a Student's t-test on the predictive accuracies of Aleph and ProGolem over the 10 folds. For the first question, since the prior expectation is that *amino acid* representation is superior to *atom-only*, we performed a one-tailed paired t-test. For the second question, since we had no prior expectation, we performed a two-tailed paired t-test.

The answer to the first question is no for Aleph (p-value = 0.195) but yes for ProGolem at the 95% confidence level (p-value = 0.015). A possible explanation as to why ProGolem takes advantage of the amino acid representation but Aleph does not is the myopia effect [KSRS97].

The myopia effect occurs because top-down ILP systems have to use a fitness function (e.g. compression) which assumes literals are conditionally independent given the target class. In domains with strong conditional dependencies between literals, such as molecular biology problems like protein-hexose binding, this approach has a poorer chance of finding good theories. Instead, a significant portion of the search resources is wasted searching very similar hypotheses.

The answer to the second question is no, i.e. we can conclude that ProGolem outperforms Aleph in both *atom-only* and *amino acid* representations. The differences in predictive accuracies between ProGolem and Aleph are statistically significant in both representations. For the

atom-only representation, the difference is statistically significant at the 95% confidence level (p-value = 0.043). For the *amino acid* representation the difference is statistically significant at the 99% confidence level (p-value = 0.004).

A third possible question would be whether ProGolem and BS kNN predictive accuracies are equivalent. Notice that this may not be a fair comparison due to the different problem representation and nature of the classifier (logical vs statistical).

A two-tailed paired t-test reveals that ProGolem outperforms BS kNN at the 99.9% confidence level (p-value = 0.0007). Such a low p-value may look surprising because it is lower than the difference between ProGolem and Aleph, despite the BS kNN mean predictive accuracy being higher than Aleph ($75.0\% \pm 6.6\%$ vs $71.3\% \pm 9.8\%$). However, the reason for such a low p-value in the paired t-test is because, for each individual fold, ProGolem predictive accuracy is always higher than BS kNN, whereas with Aleph that is not always the case.

7.3.3 Insight from rules

In this section we present the English translation and the biological explanation for some of the most compressive rules found by ProGolem using the *amino acid* representation. ProGolem rules were judged by the domain experts to be more interesting than those found by Aleph. According to ProGolem a site is hexose-binding if:

1. It contains an ASP residue whose CG atom is 5.4 ± 0.5 Å away from the binding centre, and two different ASN residues.
   [Positives covered = 37, Negatives covered = 4]

2. It contains an ASN whose N and C atoms are 2.4 ± 0.5 Å apart, and a GLU whose CB and CG atoms are 8.0 ± 0.5 Å and 6.9 ± 0.5 Å away from the binding centre, respectively.
   [Positives covered = 24, Negatives covered = 0]

3. It contains an ASN residue whose N atom is 8.2 ± 0.5 Å away from the binding centre, and an ASN residue whose N and ND2 atoms are 4.1 ± 0.5 Å apart and whose N and O

atoms are 3.6 ± 0.5 Å apart.

[Positives covered = 30, Negatives covered = 0]

4. It contains a TRP whose CB atom is 7.1 ± 0.5 Å away from the binding centre, and whose N and CD1 atoms are 4.0 ± 0.5 Å apart.

   [Positives covered = 14, Negatives covered = 0]

5. It contains a TYR whose CB and OH atoms are 5.6 ± 0.5 Å apart, a HIS whose ND1 atom is 8.9 ± 0.5 Å away from the binding centre, and a TYR whose O atom is 9.8 ± 0.5 Å away from the binding centre.

   [Positives covered = 6, Negatives covered = 0]

6. It contains CYS and LEU residues, and an ASP whose N and OD2 atoms are 4.6 ± 0.5 Å apart, and whose C atom is 7.6 ± 0.5 Å away from the binding centre.

   [Positives covered = 18, Negatives covered = 0]

The first rule requires the presence of an ASP and two ASNs. Early on, Rao et al. [RLQ98] highlighted the importance of both residues in hexose binding. Studying the lectin protein family, they report that the residues occupy identical positions independent of their sugar specificity and interact with the hexose independent of its type.

The ASP CG atom is 5.4 Å away from the centroid of the hexose pyranose ring. The pyranose radius itself being 3 Å, the ASP actually interfaces the docked hexose. Binding-site interface residues are key for hexose recognition and binding [NAAKK09], especially planar polar residues that establish a network of hydrogen bonds with the various hydroxyl groups of the docked hexose [SB04, ZSD$^+$03]. Quiocho and Vyas [QV99] report that the most common planar polar amino acids involved in hexose binding are mainly ASP and ASN, followed by GLU. ProGolem detects the role of GLU in the second rule.

The second rule also implies a triangular distance relationship between GLU's CB and CG atoms, and the binding centre. Sujatha and Balaji [SB04] report that spatial disposition of protein-galactose interacting atoms is not conserved *per se*, but is conserved with respect to

the docking position of the ligand. Similarly, ProGolem often specifies the distance of an atom with respect to the centroid of the hexose.

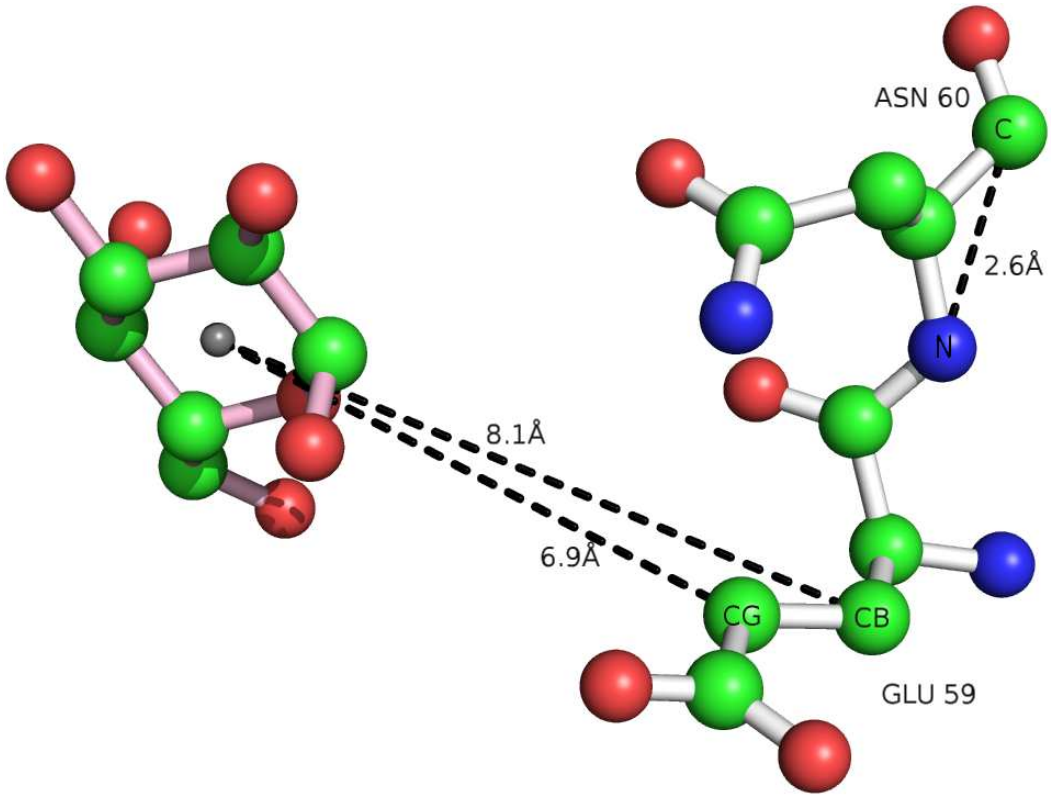


Figure 7.8: Xylanase T6. Example of rule two covering pdb id 1hiz.

Figure 7.8 is a visualization of the second rule with a particular positive example, pdb id=1hiz. In Prolog, reverse-engineering a rule to get the particular pdb ids, amino acid ids and atoms covered by a given rule is straightforward. In Figure 7.8 the ligand, a galactose, is depicted with its backbone in light pink. The two amino acids involved in the rule, a glutamic acid and an asparagine, have a white backbone. The asparagine involved in the rule has amino acid id 60 and the glutamic acid has amino acid id 59. These amino acid identifiers are relative to the pdb file for the protein 1hiz. The atoms relevant to the rule are also labelled with the actual distances for this instance of the rule shown.

In addition to specifying the distance from the binding centre, ProGolem can detect specific amino acid stereochemical dispositions. The third rule determines a particular ASN conformation, specifying the distances between backbone N and O atoms, and the side chain ND2 atom. The various spatial dispositions of the different rules need further investigation to compare

them with known 3D hexose binding-site conformations.

The aromatic residues (TRP most frequently, TYR, PHE, and to a lesser extent HIS) provide a stacking platform for the hexose to dock on [QV99]. The hexose pyranose ring forms a planar apolar hydrophobic side that stacks, through hydrophobic and van der Waals interactions, over the aromatic residues planar apolar hydrophobic side chain ring [SSKG+07]. The fourth rule requires the presence of TRP in a particular stereochemical conformation.

The fifth rule requires the presence of one or two TYR, and a HIS. This rule is thus describing a conformational representation of two or three aromatic residues around the binding-site centre. It is interesting that this low-coverage rule may indeed be capturing the infrequent sandwich interaction, whereby two or more aromatic residues engage both faces of a hexose pyranose ring [BBGD04].

The last rule asks for CYS and LEU residues. Both have negative interface propensity measures and do not form hydrogen bonds with hexoses [TJT00]. The interface propensity measure is the logarithm of the ratio between a residue frequency at the sugar binding site, and the average frequency of any residue at the binding site. A residue with a negative propensity measure does not favour the sugar binding-site region since it is present there less frequently than average. This rule covers 18 positive examples and no negative examples, and clearly specifies the presence of CYS and LEU as a discriminative factor for hexose-binding site recognition. This dependency over LEU and CYS is not previously identified in literature and merits further attention.

7.4 Conclusion

ProGolem has been developed to facilitate the learning of long, complex rules in an ILP setting. Long, complex rules are common in the molecular biology domain and we argue that a sophisticated ILP system such as ProGolem is a promising approach to automatically learn these rules from molecular data.

Our ProGolem theory has a predictive accuracy that is significantly better than previous approaches. Furthermore, by thoughtful modelling, ProGolem was able to induce rules showing a superior insight of the underlying discrimination process.

ProGolem was able to infer different aspects of the established biochemical information about hexose-binding, namely the presence of a docking aromatic residue, the importance of interface atoms, and the hydrogen-bonding activity of planar-polar residues (ASN, ASP, GLU). In addition, ProGolem was able to detect the less common aromatic sandwich interaction.

Finally, ProGolem reveals an important unreported finding: a dependency over residues CYS and LEU. It also specifies stereo configurations involving aromatic and hydrogen bonding residues. The newly reported relationship and 3D conformations require further investigation.

Chapter 8

Conclusion

In this chapter we summarize the achievements of this dissertation and suggest directions for future research.

8.1 Summary

Our **overall thesis** is that there are effective and efficient ways of searching and evaluating complex theories in an ILP setting. Over the course of this dissertation we demonstrated the overall thesis and the main contributions claimed in Section 1.1.

The ILP system we developed in Chapter 3, TopLog, introduces the Top Directed Hypothesis Derivation (TDHD) framework, an alternative way to derive hypothesis via a $\top$ theory (a logic program). Our work on TopLog already had impact on the ILP community and there is subsequent work by Dianhuan Lin [Lin09] and Domenico Corapi [CRL10] extending our approach to incorporate multi-clause learning and abduction in the TDHD framework.

Chapters 4 and 5, the core of this dissertation, addressed the overall thesis: that is, the problem of efficiently learning and evaluating long, non-determinate, predicates in ILP. The problem of learning long, non-determinate, clauses is challenging for several reasons. Current top-down methods explore only a small, relatively similar, fraction of the hypothesis space. Our ILP

system, ProGolem (see Section 4.3), performs a bottom-up beam-search on the hypothesis space using asymmetric relative minimal generalization and negative example reduction as the generalization operators.

Another challenge for learning long, non-determinate clauses, is computing the coverage of such complex clauses. In this respect we developed several coverage engines for (see Section 6.2). One of these coverage engines, Subsumer, is a general θ-subsumption engine which can be applied to a wide range of computational logic problems. Subsumer was extensively described and benchmarked in Chapter 5.

We have also shown in Sections 4.4.2 and 7.3.2 that ProGolem can be applied to a wider range of problems than Aleph, namely ones with long, non-determinate, target concepts. See in particular Section 4.4.2 where ProGolem and Aleph were applied to learn concepts from the challenging Phase Transition framework.

In Chapter 6 we have introduced GILPS, the modular and highly configurable ILP system, which implements both TopLog and ProGolem and incorporates Subsumer. The novel features introduced in GILPS, namely the sophisticated coverage engines and global theory construction with efficient cross-validation have been discussed. A tutorial showing how to use GILPS has also been presented.

Finally, in Chapter 7 we applied ProGolem to the problem of classifying whether protein sites are hexose-binding. This is a practical structural molecular biology problem, where the theory ProGolem found significantly outperformed Aleph and led to novel scientific knowledge.

We hope other researchers find the contributions of this dissertation relevant and will build upon the algorithms and systems here proposed.

8.2 Future directions

We see two main directions to continue the work presented in this disseration: improvements on the ILP framework developed (including Subsumer) and novel applications.

8.2.1 Framework improvements

It is important to incorporate new features and improvements in the GILPS framework, as these features will then be available to all ILP systems in it. A detailed discussion of the directions for future research in GILPS has been presented in Section 6.5. In summary, the main directions are: sampled clause-coverage computation, development of a standard top-down ILP engine, boosting, randomized restarts during the hypotheses search and integration of a probabilistic framework such as has been done in [LKR05].

As for improving Subsumer, one should start by implementing randomized restarts in the subsumption test. As has been shown by Resumer2 [KZ08] and our experiments in Section 5.4 randomized restarts significantly improve the performance of a subsumption engine. A different direction worth exploring is whether subsumption testing could be fully translated to other problems for which there are efficient solvers. Possible such problems are: constraint programming, sub-graph isomorphism and Boolean satisfiability (SAT).

8.2.2 Applications

It is well-known that ILP excels in relational problems [BM95] with a long history of successful application to real-world problems, e.g. [FMPS98, TMS98, KMLS92, KMSS96, SMS97].

As our experiments in Chapter 7 highlighted, a thoughtful modelling, with the help of a domain expert, is important in order to provide a good bias and a reduction of the hypothesis space. Furthermore, if the target concept is thought to be long and the background knowledge is non-determinate, we argue that it is worth employing ProGolem, as better theories may be found. In this respect it would be interesting to apply ProGolem to new and old applications of ILP where the long target concept and background knowledge non-determinacy criteria are met.

Subsumer may have applications to other areas of logic outside ILP and this should be explored. In addition to other usages reported in the literature, e.g. AI planning [Skv06], the potential use of a logic-based subsumption engine inside a Prolog compiler has come to our attention.

Appendix A

Progol Algorithms

In this appendix we present the cover-set and the most-specific clause construction algorithms of Progol. These algorithms, adapted from [Mug95b], are presented here so that the dissertation is self-contained.

Algorithm A.1 Progol's cover-set

Input: Examples E, background knowledge B, mode declarations M
Output: Theory T, a set of definite clauses

Let $T = \{\}$
Let E^+ = all positive examples in E
while E^+ contains unseen positive examples **do**
    Let e = first unseen positive example from E^+
    Mark e as seen
    Let $\bot_e$ = most-specific clause(e, B, M) (see Algorithm A.2)
    Let C_e = most compressive clause in lattice defined by $\bot_e$
    if C_e has positive score **then**
        $T := T \cup C_e$
        $E_c^+ :=$ all positive examples clause C_e covers
        $E^+ := E^+ \setminus E_c^+$
    end if
end while
return T

Algorithm A.2 Variablized most-specific clause construction

Input: Example e, background knowledge B, mode declarations M, number of variable layers i

Output: $\perp_e$, most-specific clause for e ($\perp_e = h \leftarrow b_1, \ldots, b_n$)

Let $hash$ = Hash function that uniquely maps terms to distinct logical variables
Let $InTerms = \{\}$ //Set of terms that may be used as input in a mode body declaration
Let h = the mode head declaration that subsumes e with substitution θ
for each variable/term, $v/t \in \theta$ **do**
  if v corresponds to a #type **then**
    Replace v in h by t
  end if
  if v corresponds to a +type or -type **then**
    Replace v in h by w, where w is the variable returned by $hash(t)$
    if v corresponds to a +type **then**
      $InTerms := InTerms \cup \{t\}$
    end if
  end if
end for
Set h the head of $\perp_e$
Let $CurVarDepth = 1$
while $CurVarDepth <= i$ **do**
  for each mode body declaration b **do**
    for each substitution θ of terms in $InTerms$ to input variables in b **do**
      for each solution (up to *recall* times) to goal b with answer substitution θ' **do**
        for each variable/term, $v/t, \in \{\theta \cup \theta'\}$ **do**
          if v corresponds to a #type **then**
            Replace v in b by t
          else
            Replace v in b by w, where w is the variable returned by $hash(t)$
          end if
          if v corresponds to a -type **then**
            $InTerms := InTerms \cup \{t\}$
          end if
          Add b to the body of $\perp_e$
        end for
      end for
    end for
  end for
  $CurVarDepth := CurVarDepth + 1$
end while
return $\perp_e$

Bibliography

[AK04] M. Arias and R. Khardon. Bottom-up ILP using large refinement steps. *In Proceedings of the International Conference on Inductive Logic Programming*, pages 26–43, 2004.

[BBGD04] Alisdair B. Boraston, David N. Bolam, Harry J. Gilbert, and Gideon J. Davies. Carbohydrate-binding modules: fine-tuning polysaccharide recognition. *Biochem. J.*, 382:769–781, 2004.

[BDD+02] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, 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.

[BGSS03] Marco Botta, Attilio Giordana, Lorenza Saitta, and Michèle Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

[BIA94] Henrik Boström and Peter Idestam-Almquist. Specialization of logic programs by pruning SLD-trees. In Stefan Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September*, pages 31–48. GDM-studien Nr. 237, 1994.

[BM94] Michael Bain and Stephen Muggleton. Learning optimal chess strategies. In *Machine Intelligence 13*, pages 291–309, 1994.

[BM95] I. Bratko and S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM*, 38(11):65–70, 1995.

[Bon70] Mikhail Moiseevich Bongard. *Pattern Recognition.* Spartan Books, 1970.

[BRS02] Jacques Ales Bianchetti, Céline Rouveirol, and Michèle Sebag. Constraint-based learning of long relational concepts. In Claude Sammut and Achim G. Hoffmann, editors, *ICML*, pages 35–42. Morgan Kaufmann, 2002.

[BT07] Will Bridewell and Ljupco Todorovski. Learning declarative bias. In Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli, editors, *ILP*, volume 4894 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2007.

[BWF+00] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[BZB01] R. Basilio, G. Zaverucha, and V.C. Barbosa. Learning logic programs with neural networks. *Proceedings of the International Conference on Inductive Logic Programming*, pages 15–26, 2001.

[CHH+02] Jie Cheng, Christor Hatzis, Hisashi Hayashi, Mark-A. Krogel, Shinichi Morishita, David Page, and Jun Sese. Kdd cup 2001 report. *SIGKDD Explorations*, 3(2):47–64, 2002.

[CL01] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/c̆jlin/libsvm.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill, 2001.

[CMS08] Jianzhong Chen, Stephen Muggleton, and José Carlos Almeida Santos. Learning probabilistic logic models from probabilistic examples. *Machine Learning*, 73(1):55–85, 2008.

[Coh94] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2):303–366, 1994.

[Cos09] Vítor Santos Costa. *YAP*. OPorto University, 2009. Available at http://www.dcc.fc.up.pt/∼vsc/Yap.

[CRL10] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive Logic Programming as Abductive Search. In *Technical communications of the 26th International Conference on Logic Programming, Leibniz International Proceedings in Informatics*, pages 54–63, Edinburgh, Scotland, 2010.

[CSC+03] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

[CSL07] Vítor Santos Costa, Konstantinos F. Sagonas, and Ricardo Lopes. Demand-driven indexing of Prolog clauses. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2007.

[DM86] Gerald DeJong and Raymond J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[dSC06] Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: An optimizing compiler for logic programming languages. *Journal of Universal Computer Science*, 12(7):764–787, 2006.

[DT95] Saso Dzeroski and Ljupco Todorovski. Discovering Dynamics: From Inductive Logic Programming to Machine Discovery. *Journal of Intelligent Information Systems*, 4(1):89–108, 1995.

[FMPS98] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Machine Learning*, 30:241–271, 1998.

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

[FS95] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In Paul M. B. Vitányi, editor, *EuroCOLT*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 1995.

[FW04] Marye Anne Fox and James K. Whitesell. *Organic Chemistry.* Jones & Bartlett Publishers, Boston, MA, 3rd edition, 2004.

[GS00] Attilio Giordana and Lorenza Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.

[GT07] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2007.

[HMS66] E.B. Hunt, J. Marin, and P.T. Stone. *Experiments in Induction.* Academic Press, New York, 1966.

[KCM87] Smardar T. Kedar-Cabelli and L. Thorne McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the 4th International Workshop on Machine Learning*, pages 383–389, Irvine, CA, June 1987.

[KK71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3/4):227–260, 1971.

[KL94] Jrg-Uwe Kietz and Marcus Lbbe. An Efficient Subsumption Algorithm for Inductive Logic Programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 130–138. Morgan Kaufmann, 1994.

[KMLS92] R.D. King, S.H. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89(23):11322–11326, 1992.

[KMSS96] R.D. King, S.H. Muggleton, A. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.

[KN86] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495. Springer, 1986.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In *Proceedings of IFIP Congress 74*, pages 569–574. North Holland Publishing Co., Amsterdam, 1974.

[KSRS97] Igor Kononenko, Edvard Simec, and Marko Robnik-Sikonja. Overcoming the myopia of inductive learning algorithms with RELIEFF. *Appied Intelligence*, 7(1):39–55, 1997.

[KSS95] R.D. King, A. Srinivasan, and M.J.E. Sternberg. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13:411–433, 1995.

[KZ08] Ondrej Kuzelka and Filip Železný. A restarted strategy for efficient subsumption testing. *Fundamenta Informaticae*, 89(1):95–109, 2008.

[LFZ99] Nada Lavrac, Peter A. Flach, and Blaz Zupan. Rule evaluation measures: A unifying view. In Saso Dzeroski and Peter A. Flach, editors, *ILP*, volume 1634 of *Lecture Notes in Computer Science*, pages 174–185. Springer, 1999.

[Lin09] Dianhuan Lin. Efficient, complete and declarative search in inductive logic programming. Master's thesis, Imperial College London, September 2009.

[LKFT04] Nada Lavrac, Branko Kavsek, Peter A. Flach, and Ljupco Todorovski. Subgroup discovery with cn2-sd. *Journal of Machine Learning Research*, 5:153–188, 2004.

[LKR05] Niels Landwehr, Kristian Kersting, and Luc De Raedt. nfoil: Integrating naïve bayes and foil. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 795–800. AAAI Press / The MIT Press, 2005.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[LM77] J. Larson and R. S. Michalski. Inductive inference of vl decision rules. *SIGART Bulletin*, 63:38–44, 1977.

[MF92] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, London, 1992.

[MF01] S.H. Muggleton and J. Firth. *CProgol4.4: A Tutorial Introduction*, pages 160–188. Springer-Verlag, 2001.

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

[MKS92] S.H. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[MRSV06] Pierre Mahé, Liva Ralaivola, Véronique Stoven, and Jean-Philippe Vert. The pharmacophore kernel for virtual screening with support vector machines. *J. Chem. Inf. Model.*, 46(5):2003–2014, 2006.

[MS04] Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

[MSTN08] Stephen Muggleton, José Carlos Almeida Santos, and Alireza Tamaddoni-Nezhad. TopLog: ILP using a logic program declarative bias. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 687–692. Springer, 2008. An extended version of this paper is available at http://www.doc.ic.ac.uk/∼jcs06/TopLog.

[MSTN09] Stephen Muggleton, José Santos, and Alireza Tamaddoni-Nezhad. ProGolem: a system based on relative minimal generalisation. In *Proceedings of the 19th International Conference on ILP, Springer*, pages 131–148, Leuven, Belgium, 2009.

[MTN07] S.H. Muggleton and A. Tamaddoni-Nezhad. QG/GA: A stochastic search for Progol. *Machine Learning*, 70(2–3):123–133, 2007.

[Mug95a] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[Mug95b] Stephen Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13(3&4):245–286, 1995.

[NAAK+09] Houssam Nassif, Hassan Al-Ali, Sawsan Khuri, Walid Keirouz, and David Page. An Inductive Logic Programming approach to validate hexose biochemical knowledge. In *Proceedings of the 19th International Conference on ILP*, pages 149–165, Leuven, Belgium, 2009.

[NAAKK09] Houssam Nassif, Hassan Al-Ali, Sawsan Khuri, and Walid Keirouz. Prediction of protein-glucose binding sites using Support Vector Machines. *Proteins*, 77(1):121–132, 2009.

[NCdW97] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

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

[PS03] David Page and Ashwin Srinivasan. ILP: A short look back and a longer look forward. *Journal of Machine Learning Research*, 4:415–430, 2003.

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

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[QV99] F. A. Quiocho and N. K. Vyas. Atomic interactions between proteins/enzymes and carbohydrates. In S. M. Hecht, editor, *Bioorganic Chemistry: Carbohydrates*, chapter 11, pages 441–457. Oxford University Press, New York, 1999.

[RB93] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th IJCAI*. Morgan Kaufmann, 1993.

[RFKM08] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.

[RLQ98] V. S. R. Rao, K. Lam, and P. K. Qasba. Architecture of the sugar binding sites in carbohydrate binding proteins—a computer modeling study. *International Journal of Biological Macromolecules*, 23(4):295–307, 1998.

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

[RW00] A.M. Richard and C.R. Williams. Distributed structure-searchable toxicity (DSSTox) public database network: A proposal. *Mutation Research*, 499:27–52, 2000.

[San10] José Carlos Almeida Santos. *General Inductive Logic Programming System*. Imperial College, London, 2010. Available at http://ilp.doc.ic.ac.uk/GILPS.

[SB04] M. S. Sujatha and P. V. Balaji. Identification of common structural features of binding sites in galactose-specific proteins. *Proteins*, 55(1):44–65, 2004.

[SBM07] E. Solomon, L. Berg, and D. W. Martin. *Biology*. Brooks Cole, Belmont, CA, 8th edition, 2007.

[SG85] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[SHW96] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient Theta-Subsumption Based on Graph Algorithms. In Stephen Muggleton, editor, *Inductive Logic Programming Workshop*, volume 1314 of *Lecture Notes in Computer Science*, pages 212–228. Springer, 1996.

[Skv06] Olga Skvortsova. θ-subsumption based on object context. In Stephen Muggleton, Ramón P. Otero, and Alireza Tamaddoni-Nezhad, editors, *ILP*, volume 4455 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2006.

[SM10a] José Santos and Stephen Muggleton. Subsumer: A Prolog theta-subsumption engine. In *Technical Communications of the 26th International Conference on Logic Programming, Leibniz International Proceedings in Informatics*, pages 172–181, Edinburgh, Scotland, 2010.

[SM10b] José Santos and Stephen Muggleton. When does it pay off to use sophisticated entailment engines in ILP? In *Proceedings of the 20th International Conference on ILP*, Florence, Italy, 2010. Springer. In press.

[SMS97] A. Srinivasan, R.D. King S.H. Muggleton, and M. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on ILP*, pages 273–287. Springer-Verlag, Berlin, 1997. LNAI 1297.

[SR97] Michèle Sebag and Céline Rouveirol. Tractable induction and classification in first order logic via stochastic matching. In *IJCAI (2)*, pages 888–893, 1997.

[Sri07] Ashwin Srinivasan. *The Aleph Manual*. University of Oxford, 2007. Available at http://web2.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html.

[SSKG+07] James Screen, E. Cristina Stanca-Kaposta, David P. Gamblin, Bo Liu, Neil A. Macleod, Lavina C. Snoek, Benjamin G. Davis, and John P. Simons. IR-spectral signatures of aromaticsugar complexes: Probing carbohydrateprotein interactions. *Angew. Chem. Int. Ed.*, 46:3644–3648, 2007.

[STNM09] José Carlos Almeida Santos, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. An ILP system for learning head output connected predicates. In Luís Seabra Lopes, Nuno Lau, Pedro Mariano, and Luís Mateus Rocha, editors, *EPIA*, volume 5816 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2009.

[TJT00] C. Taroni, S. Jones, and J. M. Thornton. Analysis and prediction of carbohydrate binding sites. *Protein Eng.*, 13(2):89–98, 2000.

[TMS98] M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. Protein fold recognition. In C. D. Page, editor, *Proceedings of the 8th International Workshop on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 53–64. Springer-Verlag, Berlin, 1998.

[TNM09] A. Tamaddoni-Nezhad and S.H. Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76(1):37–72, 2009.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Whi80] P. Whittle. Multi-Armed Bandits and the Gittins Index. *Journal of the Royal Statistical Society. Series B (Methodological)*, 42(2):143–149, 1980.

[ZSD+03] Y. Zhang, G. J. Swaminathan, A. Deshpande, E. Boix, R. Natesh, Z. Xie, K. R. Acharya, and K. Brew. Roles of individual enzyme-substrate interactions by alpha-1,3-galactosyltransferase in catalysis and specificity. *Biochemistry*, 42(46):13512–13521, 2003.

[ZSJ06] Filip Železný, Ashwin Srinivasan, and C. David Page Jr. Randomised restarted search in ILP. *Machine Learning*, 64(1-3):183–208, 2006.