The Dissertation Committee for Lap Poon Rupert Tang

certifies that this is the approved version of the following dissertation:

Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining

Committee:

Raymond Mooney, Supervisor

Gordon Novak

Bruce Porter

Risto Miikkulainen

David Page

Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining

by

Lap Poon Rupert Tang, B.S.,M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2003

To my father, the force behind my work, if only he could share with me the joy ...

To my mother who always believes in me even when I don't.

Also to Christ Jesus without whom the days would have been much harder if possible.

Acknowledgments

The completion of this thesis hinges on many factors. The most important of which would be the advice, besides patience, of my supervisor Dr. Raymond J. Mooney. I would like to take this opportunity to thank Prem Melville for his assistance on data preparation. He has been a valuable partner on the EELD project and a very helpful friend in our research group. My thanks also goes to Sugato Basu, my officemate and friend, who has been a valuable companion in the course of completing my thesis. We had many stimulating discussions on various topics in Artifical Intelligence, particularly in Machine Learning. He's an interesting person to talk to and share research ideas with.

Much of thanks go to Vitor Santos Costa who has gone many extra miles to help with my development of the Prolog code for BETH and in running ALEPH on the EELD data besides many other advices and help offered to our research group on the EELD project like installing the Yap Prolog compiler for our department. Also, I want to thank James A. Bednar for his assistance in my course of preparing this document. Harold Chaput has also offered timely help. I also want to thank Gloria Ramirez for her help on numerous occasions from resolving tuition issues to getting my forms signed. They were many other important pieces of assistance which helped bring my work to a fruition. I would like to convey my gratitude to these people albeit their names have already slipped away from memory.

Special thanks also go to the people in the DaimlerChrysler Research and Technology Center who made the experimental database in the Northern California restaurant domain available for research purposes. The research was supported by the National Science Foundation under grant IRI-9704943 and a grant from the DaimlerChrysler Research and Technology Center at Palo Alto in California.

This research is sponsored by DARPA and managed by Rome Laboratory under contract F30602-01-2-0571. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

LAP POON RUPERT TANG

The University of Texas at Austin

December 2003

Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining

Publication No. ____________________

Lap Poon Rupert Tang, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Raymond Mooney

Inductive Logic Programming (ILP) is the intersection of Machine Learning and Logic Programming in which the learner's hypothesis space is the set of logic programs. There are two major ILP approaches: top-down and bottom-up. The former searches the hypothesis space from general to specific while the latter the other way round. Integrating both approaches has been demonstrated to be more effective. Integrated ILP systems were previously developed for two tasks: learning semantic parsers (Chillin), and mining relational data (Progol). Two new integrated ILP systems for these tasks that overcome limitations of existing methods will be presented.

Cocktail is a new ILP algorithm for inducing semantic parsers. For this task, two features of a parse state, functional structure and context, provide im-

portant information for disambiguation. A bottom-up approach is more suitable for learning the former, while top-down is better for the latter. By allowing both approaches to induce program clauses and choosing the best combination of their results, COCKTAIL learns more effective parsers. Experimental results on learning natural-language interfaces for two databases demonstrate that it learns more accurate parsers than CHILLIN, the previous best method for this task.

BETH is a new integrated ILP algorithm for relational data mining. The Inverse Entailment approach to ILP, implemented in the PROGOL and ALEPH systems, starts with the construction of a *bottom clause*, the most specific hypothesis covering a seed example. When mining relational data with a large number of background facts, the bottom clause becomes intractably large, making learning very inefficient. A top-down approach heuristically guides the construction of clauses without building a bottom clause; however, it wastes time exploring clauses that cover no positive examples. By using a top-down approach to heuristically guide the construction of generalizations of a bottom clause, BETH combines the strength of both approaches. Learning patterns for detecting potential terrorist activity is a current challenge problem for relational data mining. Experimental results on artificial data for this task with over half a million facts show that BETH is significantly more efficient at discovering such patterns than ALEPH and M-FOIL, two leading ILP systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning concerns with the question of how to construct computer programs that automatically improve with experience (a.k.a. learning algorithms) (Mitchell, 1997). Many a successful machine learning application has been developed in the past three, perhaps even five, decades of research, ranging from programs that can automatically translate a piece of text in one language to another, to programs that can read aloud a given piece of text, e.g. in English, to aid blind people, to data-mining programs that learn to detect fraudulent credit card transactions.

1.1 Attribute Value Learning and Limitations

Earlier work in machine learning focused on tasks that involve the learning of classification functions from data represented by a vector of attributes and their values (a.k.a. attribute-value representation) (Michalski, 1983; Quinlan, 1986; Rumelhart, Hinton, & Williams, 1986). For example, a system could learn to classify face images, represented as bitmaps, based on which person was pictured. Another example would be the learning of association rules that predict the products a customer would purchase (a.k.a. purchasing behavior) based on the customer's "attributes" like gender, age, career, hobbies, and, perhaps, even the customer's prior purchasing behavior. Most of these earlier work can be summarized under the title "attribute value learning" in which the representation of an object (i.e. training example) is in the form of a vector of values (one for each attribute). One popular approach is learning with neural networks in which the attribute values are numerical. Yet, another popular approach is "symbolic machine learning" in which attribute values can be either numerical or simply a symbol (i.e. a sequence of alphanumerical characters). Attribute value learning in this framework has come to be called propositional learning since these systems find expressions equivalent to sentences in propositional logic.

Fortunately or unfortunately, as the nature of problems becomes more sophisticated, so is the expressiveness (of the learning system) required to capture and represent objects or concepts to be learned. We are going to focus on tasks that require a more complicated representation framework than that of propositional logic. More precisely, we will focus on two tasks which require something not offered in propositional logic. The first one is semantic parser acquistion which requires the use of function terms since the state of a parser is most conveniently represented as a tree structure of various objects like the *list* of words in the input buffer and so on (Zelle & Mooney, 1993). The second task is data mining with multi-relational databases which requires the use of variables and predicates to capture relational knowledge embedded in the relational tables (Džeroski & Lavrač, 2001). Unfortunately, none of these features is available in propositional logic.

1.2 Inductive Logic Programming

The next level of expressiveness in terms of the power of representation is offered by the framework in first-order logic. However, even first-order logic is more expressive than necessary. Only a subset of first-order logic that is commonly called Horn clause first-order logic, in which sentences are in the form of "if-then rules" for more efficient inference, is already sufficient for our purpose. Fortunately, an effective mechanism of its computation has been formulated in the well-developed framework of SLDNF-resolution in logic programming [1] (Doets, 1994) and implemented in PROLOG (Bratko, 1990).

Inductive logic programming (ILP) is the intersection of machine learning and logic programming, which concerns developing learning algorithms within the framework of Horn clause first-order logic (Lavrac & Dzeroski, 1994a). The past decade of research in ILP has produced two major approaches: 1) top-down and 2) bottom-up. The former is characterized by searching the hypothesis space in a general to specific manner (Quinlan, 1990) while the latter in a specific to general order (Bain & Muggleton, 1992). Bottom-up approaches were pioneered by Stephen Muggleton who started the first workshop for Inductive Logic Programming in 1991, which became popular within the machine learning community in Europe and subsequently the ESPRIT (Ecoles) Project [2] started with a view to promoting the development of techniques and applications in ILP. Quite a few useful ILP systems were developed under this project, which were also applied to many interesting areas. One of the more interesting applications was, perhaps, in the now

[1] Selection-rule driven Linear Resolution for Definite clauses extended with Negation as Failure
[2] The latest "descendant" is the ESPRIT IV Project.

rapidly growing decipline called bioinformatics that concerns machine aided drug design (Finn, Muggleton, Page, & Srinivasan, 1998). The top-down approach was first started by Ross Quinlan who developed the well known learning system C4.5 (Quinlan, 1986), from which the basic approach was extended to handling Horn clause logic in the system called FOIL (Quinlan, 1990) that was also applied to many interesting research problems like text categorization, for instance. Today, ILP has already become a well established decipline within machine learing with many different areas of useful applications, even started merging with other areas of research like probabilistic reasoning (Kersting & Raedt, 2001).

1.3 Integrating Top-down and Bottom-up Approaches in ILP

The most popular top-down approach in ILP has to be FOIL (or mFOIL) while GOLEM probably represents the best bottom-up approach in ILP. There were two earlier ILP algorithms based on the idea of combining top down and bottom up approaches: 1) CHILLIN (Zelle & Mooney, 1994), which is based largely on combining GOLEM and FOIL, and 2) PROGOL (Muggleton, 1995), which is based on combining FOIL with a framework called Inverse Entailment (originated from bottom-up approaches). They represented the best "ILP hybrids" to the author's knowledge; the former was applied to semantic parser acquisition (Zelle & Mooney, 1993) and the latter to pharmacophore discovery in bioinformatics (Finn et al., 1998) and other tasks as well. However, each of them has their own shortcomings. More precisely, CHILLIN's particular way of combining GOLEM and FOIL makes it difficult to utilize background knowledge that would normally/intuitively be given to a typical top-down ILP system, which negatively affected the *accuracy* achieved on the task. PROGOL has to construct a (huge) bottom clause before starting to look for a hypothesis in a FOIL-like manner, which makes learning *inefficient* in domains with a large number of facts in the background knowledge. In this thesis, we are going to present more advanced techniques on combining top-down and bottom-up approaches, one for each type of domains, that can overcome these various shortcomings encountered in these earlier ILP hybrids.

1.4 Application of ILP in Semantic Parser Acquisition

One of the goals in Artificial Intelligence (AI) is the developement of systems with abilities of processing natural languages like that of human beings, which spawned a sub-field in AI called Natural Language Processing (NLP). The more well known tasks in NLP among many others are parsing and speech recognition. Parsing can be broadly and very roughly divided into two areas: 1) syntactic parsing and 2) semantic parsing. The former emphasizes recognizing and manipulating the syntactic structures of natural language sentences while the latter the meaning or possible interpretations of natural language sentences. We will only focus on the latter in this thesis.

Semantic parsing refers to the process of mapping a natural language input (a sentence) to some structured meaning representation which is suitable for manipulation by a machine (Allen, 1995). For example, in building a natural language interface (NLI) for a commercial database, one may want to map a user data request expressed in a natural language to the underlying database access language like SQL. The target query which is expressed in SQL, in this case, would serve as the meaning representation for the user request. The choice of a semantic representation language is entirely domain dependent since as of now there has not been developed a "univeral" semantic representation language which is expressive enough to handle the world of possible meaning structures. Semantic parsing is a difficult problem in natural language processing (NLP) since anyone who attempts to approach it would necessarily have to tackle the very difficult task of *natural language understanding.*

Semantic parsing has been an interesting problem in NLP as it would very likely be part of any interesting NLP applications, particularly those that would require translation of a natural language input to a specific command. Research in semantic parsing with a focus on developing natural language interfaces for database querying started in the 70's (Woods, 1970; Waltz, 1978) and carries on to the 90's (Miller, Stallard, Bobrow, & Schwartz, 1996; Zelle, 1995; Kuhn & De Mori, 1995). With the advent of the "information age", the availabilty of such applications would definitely widen the "information delivery bottleneck". Online database access in natural languages makes information available to users who do not necessarily possess the knowledge of the underlying database access language and therefore makes information a lot more accessible. One great potential impact would be on the utility of the World Wide Web where information could be delivered through NLIs implemented as Web pages. The success of semantic parsing techiques would definitely be the cornerstone of a prospering development of such interesting applications. [3]

There has been a resurgence of empirical approaches to natural language

[3] Even though building NLIs to databases is being emphasized here, we by no means imply that it would be the only important application of semantic parsing.

processing since the late 1980's. The success of such approaches in areas like speech recognition (Rabiner, 1989; Bahl, Jelinek, & Mercer, 1983), part-of-speech tagging (Charniak, Hendrickson, Jacobson, & Perkowitz, 1993), syntactic parsing (Ratnaparkhi, 1999; Manning & Carpenter, 1997; Charniak, 1996; Collins, 1997; Pereira & Shabes, 1992), and text or discourse segmentation (Litman, 1996) is evidential. In fact, it has been coined a "revolution" within the NLP community (Hirschberg, 1998). There are reasons why such approaches have experienced a resurgence: 1) the success in information and networking technology has made large volumes of real world corpora of text available (which could serve the role of "raw data" in any empirical approach) and 2) empirical approaches have proven successful to develop systems that satisfy some of the desirable properties of an NLP application, namely a) *acquisition*, automatically acquiring knowledge (domain specific or not) that would be necessary for the task, b) *coverage*, handling the potentially wide range of possibilities that could arise in the application, c) *robustness*, accommodating real data which may not be "perfect" (like having noise) and still being able to perform reasonably well, d) *portability*, easily applicable to a different task in a new domain (Armstrong-Warwick, 1993). A system called CHILL, which employed an empirical approach to semantic parser acquisition, was developed and demonstrated to perform better than hand-crafted parsers in several domains (Zelle & Mooney, 1993).

One of the factors, perhaps the largest one, affecting the performance of a system like CHILL is the quality of the induction algorithm embedded in it. In CHILL, we used an induction algorithm called CHILLIN, which has the problem mentioned before. We developed a new ILP algorithm we call COCKTAIL, which is based also on the idea of integrating top-down and bottom-up approaches in ILP like CHILLIN. However, unlike CHILLIN which, perhaps, put too much weight on the bottom-up approach than the top-down approach making it difficult to utilize background knowledge, COCKTAIL gives equal weight to each type of ILP approach. Each ILP algorithm employed, one from each of the ILP approaches, in the system retains its respective strength. Significant improvement in CHILL's performance is achieved using such a new ILP algorithm.

1.5 Application of ILP in Relational Data Mining

Knowledge discovery in databases (KDD) is the (non-trivial) process of identifying valid, novel, potentially useful, and understandable patterns in data (Džeroski & Lavrač, 2001). Data mining (DM) is, perhaps, the "central" step in the KDD process, which concerns applying computational techniques (in the form of a learning algorithm) to actually find patterns in the data. Other steps involve preparation of data (maybe as well as collection of data) and evaluation of discovered patterns.

Most existing data mining approaches look for patterns in a single table of data (Agrawal, Imielinski, & Swami, 1993). *Relational data mining* approaches, on the other hand, look for patterns that involve multiple relations from a relational database (Finn et al., 1998). The data taken as input by these approaches thus typically consists of several tables and not just one table. ILP is particularly suitable for learning relational knowledge from these relational databases due to the ease with which relational information is represented under the framework of Horn clause first-order logic. We will look at a novel kind of relational data mining called link discovery in which interesting patterns are learned from a very large relational database with many tables and many relations represented as Prolog facts.

The terrible events of September 11, 2001 have sparked increased development of information technology that can aid intelligence agencies in detecting and preventing terrorism. The Evidence Extraction and Link Discovery (EELD) program of the Defense Advanced Research Projects Agency (DARPA) is one attempt to develop new computational methods for addressing this problem. More precisely, *Link Discovery* (LD) is the task of identifying known, complex, multi-relational patterns that indicate potentially threatening activities in large amounts of relational data. Some of the input data for LD comes from *Evidence Extraction* (EE), which is the task of obtaining structured evidence data from unstructured, natural-language documents (e.g. news reports), other input data comes from existing relational databases (e.g. financial and other transaction data). Finally, *Pattern Learning* (PL) concerns the automated discovery of new relational patterns for detecting potentially threatening activities in large amounts of multi-relational data.

PROGOL (Muggleton, 1995) is considered to be the state-of-the-art ILP algorithm in tackling problems in relational data mining. It was applied on various problems in bioinformatics which was the largest ILP problem before link discovery to the author's knowledge. As we shall see, link discovery has an unprecedented explosion in the amount of data, in terms of the total number of background facts. The problem is so large that state-of-the-art ILP algorithms become inefficient in computation. Like we mentioned, PROGOL constructs a huge bottom clause, which makes the search space of hypotheses unnecessarily large. We will present our new approach (also based on integrating the two approaches in ILP) we call BETH that was designed to specifically address this issue. We will experimentally demonstrate that such a new approach can significantly improve the efficiency of link discovery.

1.6 Reducing Complexity of Theorem Proving in ILP

Every ILP system has to tackle the problem of theorem proving in Horn clause logic since finding the set of positive and negative training examples covered by a clause is the prerequisite to computing the heuristic value of the clause (e.g., m-estimate or statistical significance), which is an essential part in the search for a good clause. However, simply proving an ordinary clause takes much time as theorem proving is an NP hard problem in general (Cook, 1971). Query transformation (Costa, Srinivasan, Camacho, Blockeel, B.Demoen, Janssens, Struyf, Vandecasteele, & Laer, 2002) concerns developing techniques that transform a given clause by appropriately inserting cut operators (!) and $once/1$ predicates so that the resulted clause becomes much easier to prove. We are going to overview two state-of-the-art techniques called cut-transformation and once-transformation before we proceed to present our own algorithm based on these techniques we call *incremental* cut-and-once transformation that is theoretically more efficient. Although we did not actually employ our own algorithm in our ILP systems for the sake of simplicity in the implementation of BETH, we present the idea as a theoretical result (or contribution) in the thesis.

1.7 Organization of Thesis

Two ILP hybrids, CHILLIN and PROGOL, have been applied to semantic parser acquisition and problems in relational data mining respectively. Each of them has its own shortcomings; the former negatively affected the accuracy of the parser while the latter led to inefficiency in learning. In this thesis, we are going to present two more advanced techniques re combining top-down and bottom-up approaches in ILP, namely COCKTAIL and BETH, which can overcome the shortcomings of these earlier ILP hybrids.

The rest of the thesis is organized as follows. Chapter 2 will give a brief introduction to Inductive Logic Programming. An empirical approach to NLP called CHILL will be overviewed in Chapter 3. The new ILP algorithm COCKTAIL developed for further improving CHILL's performance is given in Chapter 4. In Chapter 5, we will address the data mining problem called link discovery as mentioned and present our new ILP algorithm that substantially boosted the efficiency of link discovery. After that, we will give an introduction to query transformation, from which we employed the techniques in our ILP systems, along with our theoretical contribution (the incremental cut-and-once transformation) in Chapter 6. We will then briefly review some related work or research problems in Chapter 7 followed by a discussion on possible future research work in Chapter 8. Finally, we will present our conclusions in Chapter 9.

Chapter 2

Background on Inductive Logic Programming

2.1 Preliminaries

We will start by giving some basic concepts and definitions from the theory of logic programming. Please refer to (Lloyd, 1984) for more details. We assume basic knowledge of first-order logic and its vocabulary.

A *clause* is a set of literals. A positive literal is an un-negated literal. For example, $l(A, B)$ is a positive literal. A negative literal is a literal preceded by the classical negation. For example $\neg l(A, B)$ is a negative literal. Some use an overline to represent classical negation. So, $\overline{l(A, B)}$ is another way to represent the negative literal $\neg l(A, B)$.

A *Horn clause* C is a set of literals (implicitly representing a disjunction) with at most one positive literal and therefore it takes the form: $H \leftarrow B$ where H, the *head* of C, is a literal and B, the *body* of C, is a set of literals. For example, $D : f(A,B) \leftarrow p(B,C), q(C,A)$ is a Horn clause. However, D can also be written as this set of literals: $\{f(A,B), \overline{p(B,C)}, \overline{q(C,A)}\}$.

Given a clause $H \leftarrow B$, the set of variables in H are called *head variables*. For example, the set of head variables for the clause $f(A,B) \leftarrow p(B,C), q(C,A)$ is $\{A, B\}$.

The *close world assumption* says that if a literal L cannot be inferred from a given set of background knowledge BK, then L is not satisfiable under BK, i.e. $BK \not\models L$.

Negation as Failure (denoted as *not*) is a form of negation implemented in Prolog based on the close world assumption. It is logically different from classical negation $\neg$.

A Horn clause is *definite* if there is no negation as failure *not* nor classical negation $\neg$ in the body of the clause. For example, given two clauses $C_1 : H \leftarrow L_1, L_2$ and $C_2 : H \leftarrow notL_1, L_2$, only C_1 is a definite clause.

A Horn clause is *range restricted* if and only if its head variables all appear in the body of the clause. For example, $f(A,B) \leftarrow p(B,C), q(C,A)$ is a range restricted clause but $f(A,B) \leftarrow p(B,C), q(C,D)$ is not a range restricted clause.

A Horn clause is *function free* if there exists no function terms anywhere in the clause. For example, $f(A,B) \leftarrow p(B,C), q(C,A)$ is a function free clause but $f(g(A),B) \leftarrow p(B,C), q(C,A)$ is not a function free clause because the function term $g(A)$ appears in the head of the clause.

A *substitution* is a set of bindings of the form: V/c where c is a constant and V is a variable. For example, $\{A/a, B/b\}$ is a substitution which binds the variable

A to the constant a and the variable B to the constant b.

We use the notation $T\theta$ to denote the result of binding all the variables in the term T according to the substitution θ. For example, suppose $T = f(A, B)$ and $\theta = \{A/a, B/b\}$, then $T\theta = f(a, b)$.

A clause C θ-subsumes ($\preceq$) a clause D if there exists substitution θ such that $C\theta \subseteq D$. For example, the clause $C : f(A, B) \leftarrow p(B, G), q(G, A)$ θ-subsumes the clause $D : f(a, b) \leftarrow p(b, g), q(g, a), t(a, d)$ by the substitution $\{A/a, B/b, G/g\}$. Therefore, $C \preceq D$.

A *solution* to a literal L is a substitution θ such that the given background knowledge $BK \models L\theta$. For example, if the given set of background knowledge is $BK = \{p(a_1, b_1), p(a_2, b_2), p(a_3, b_3)\}$, the literal $p(X, Y)$ has three solutions: $\{X/a_1, Y/b_1\}$, $\{X/a_2, Y/b_2\}$, and $\{X/a_3, Y/b_3\}$.

A substitution θ *satisfies* a set of literals S for a given set of background knowledge BK if $BK \models S\theta$.

The Prolog cut (!) is a procedural operator used to prevent backtracking to avoid unnecessary non-determinacy.

The followings are some basic definitions from the literature in Inductive Logic Programming such as (Lavrac & Dzeroski, 1994a) and (Muggleton, 1995).

The *target predicate* is the predicate of the concept that one wants to learn. For example, in learning the concept $grandparent(X, Y)$ (X is the grandparent of Y), $grandparent$ is the target predicate.

Given a clause $H \leftarrow B$, the *depth* of a variable V (*variable depth*) $d(V)$ is defined as: 1) for each V which appears in H, $d(V) = 0$ and 2) From left to right, for each literal $L \in B$, if V_{max} is the variable with the maximum known depth in L, then every variable V_i in L whose depth is to be calculated, $d(V_i) = d(V_{max}) + 1$. For example, given the clause $f(A,B) \leftarrow g(A,D), p(A,D,E), q(E,R,A,B)$, the variable depth of $d(A) = 0, d(B) = 0, d(D) = 1, d(E) = 2, d(R) = 3$.

The *variable depth bound* on a given clause is the maximum depth a variable can have for any variable which appears in the clause.

Recall bound is the maximum number of alternative solutions of a predicate L.

The *clause length* is the total number of literals in the body of the clause.

A set of background knowledge is defined *extentionally* if it is represented as a set of ground literals like $\{grandparent(Bob, Susan), grandparent(Tom, Mary)\}$. It is defined *intentionally* if it is represented as a Horn clause theory with quantified variables. For example, $\{grandparent(X,Y) \leftarrow parent(X,Z), parent(Z,Y)\}$ and a set of grounded facts (grounded literals) defining the parent relationship serve as an intentionally defined set of background knowledge.

2.2 Problem Definition

Inductive Logic Programming (ILP) is a growing subfield in AI at the intersection of machine learning and logic programming. The problem is defined as follows. Given a set of examples (represented as ground literals) $\xi = \xi^+ \cup \xi^-$ consisting of positive and negative examples of a target concept, and background knowledge B, find an hypothesis $H \in \mathcal{L}$ (the language of hypotheses) such that the following conditions hold [1] (Muggleton & Raedt, 1994).

Prior Satisfiability. $B \wedge \xi^- \not\models \Box$

[1] This problem setting is also called the *normal semantics* of ILP.

Posterior Satisfiability. $B \wedge H \wedge \xi^- \not\models \Box$

Prior Necessity. $B \not\models \xi^+$

Posterior Sufficiency. $B \wedge H \models \xi^+$

where $\Box$ represents logical falsity (a contradiction). The sufficiency criterion is also called *completeness* with regard to positive examples and the posterior satisfiability criterion is also known as *consistency* with the negative examples. Due to the use of a more expressive first-order formalism, ILP techniques are proven to be more effective in tackling problems that require learning relational knowledge than traditional propositional approaches (Quinlan, 1990).

There are two major approaches in the design of ILP learning algorithms: top-down and bottom-up. Both approaches can be viewed more generally as a kind of *set covering algorithm*. However, they differ in the way a clause is constructed. In a top-down approach, one builds a clause in a general to specific order where the search usually starts with the most general clause and successively specializes it with background predicates according to some search heuristic. A representative example of this approach would be the FOIL algorithm (Quinlan, 1990; Cameron-Jones & Quinlan, 1994). In a bottom-up approach, the search begins at the other end of the space where it starts with the most specific hypothesis, the set of examples, and constructs the clauses in a specific to general order by generalizing the more specific clauses. We will briefly review a number of earlier ILP systems based on this approach.

2.3 Top Down Approach

Top-down ILP methods learn programs by generating clauses one after the other, and generate clauses by means of specialization. We are going to review two popular systems that have been developed in the past decade of research in ILP.

2.3.1 FOIL and mFOIL

FOIL (Quinlan, 1990) may be one of the earliest top-down rule learning algorithms which use a hypothesis space more expressive than that of traditional propositional rule learning algorithms like, for instance, C4.5 rules (Quinlan, 1993). More precisely, FOIL learns a function-free first-order Horn clause definition of a target concept given the *background predicates* which are defined extensionally. FOIL contains an outer loop, which in each iteration, finds a clause that covers a portion of the positive examples and are consistent with the negative examples. The loop stops when the set of clauses found covers all the positive examples. The inner loop builds a single clause, starting with the most general hypothesis and adding literals to it until it covers no negative examples. Literals are ranked using the information gain metric and the literal that maximizes gain is chosen. More formally, let T_+ denote the number of positive tuples in the set T, the information of T is defined as:

$$I(T) = -\log_2(T_+/\mid T\mid). \tag{2.1}$$

And, the information gain of a literal L is defined as:

$$Gain(L) = s \cdot (I(T) - I(T')) \tag{2.2}$$

where s is the number of tuples in T that have extensions in T' (i.e. the number of current positive tuples covered by L) and T' is the new training set created from T

and L. Figure 2.1 summarizes the FOIL algorithm.

Like FOIL, mFOIL (Lavrac & Dzeroski, 1994b) is a top-down ILP algorithm. However, it uses a more direct accuracy estimate, the m-estimate (Cestnik, 1990), to measure the expected accuracy of a clause which is defined as

$$accuracy(C) = \frac{s + m \cdot p^{+}}{n + m} \tag{2.3}$$

where C is a clause, s is the number of positive examples covered by the clause, n is the total number of examples covered, p^{+} is the prior probability of the class $\oplus$, and m is a parameter.

mFOIL was designed with handling imperfect data in mind. It uses a pre-pruning algorithm which checks if a refinement of a clause can be *possibly* significant. If so, it is retained in the search. The significant test is based on the likelihood ratio statistic. Suppose a clause covers n examples, s of which are positive examples, the value of the statistic is calculated as follows:

$$\textit{Likelihood Ratio} = 2n(q^{+} \log \frac{q^{+}}{p^{+}} + q^{-} \log \frac{q^{-}}{p^{-}}) \tag{2.4}$$

where p^{+} and p^{-} are the prior probabilities of the class $\oplus$ and $\ominus$ respectively, $q^{+} = \frac{s}{n}$, and $q^{-} = 1 - q^{+}$. This is distributed approximately as χ^2 with 1 degree of freedom. If the estimated value of a clause is above a particular threshold, it is considered significant. A clause, therefore, cannot be possibly significant if the upper bound $-2s \log p^{+}$ is already less than the threshold, and will not be further refined.

The search starts with the most general clause. Literals are added successively to the body of a clause. A beam of promising clauses are maintained, however, to partially overcome local minima. The search stops when no clauses in the beam can be significantly refined and the most significant one is returned.

Procedure *Foil*
Input:
$R(V_1, V_2, ..., V_k)$: the target concept
ξ^+: the $\oplus$ examples
ξ^-: the $\ominus$ examples
Output:
H: the set of learned clauses

$H := \emptyset$
Positives-To-Cover $:= \xi^+$
While *Positives-To-Cover* is not empty **Do**
    /* Search for a consistent clause covering a preferably large subset of *Positives-To-Cover*.*/
    $C := R(V_1, V_2, ..., V_k) \leftarrow$
    $T :=$ *Positives-To-Cover* $\cup\ \xi^-$
    While T contains negative tuples **Do**
        Find the literal L that maximizes $Gain(L)$ to add to the clause C
        Form a new set T' by extending each tuple t in T that satisfies L
        with its new variable bindings
    Replace T by T'
    End While
    Add C to H
    Remove examples covered by C from *Positives-To-Cover*
End While
Return H
End Procedure

Figure 2.1: The FOIL algorithm

2.4 Bottom Up Approaches

Bottom up approaches are all based on the generalization of positive examples through *generalization operators.* We will briefly discuss a number of earlier bottom up ILP systems.

2.4.1 Least General Generalization

Plotkin's least general generalization (LGG) (Plotkin, 1970) was perhaps the first formal analysis on the notion of generalization as a process of inductive inference.

The LGG of two terms $f_1(l_1, ..., l_n)$ and $f_2(m_1, ..., m_n)$ is a new variable v if $f_1 \neq f_2$. Otherwise, it is $f_1(lgg(l_1, m_1), ..., lgg(l_n, m_n))$. The LGG of two literals $L_1 = (\neg)p(t_1, \dots, t_n)$ and $L_2 = (\neg)q(s_1, \dots, s_n)$ is undefined if L_1 and L_2 do not have the same predicative symbol and sign; otherwise, it is defined as:

$$lgg(L_1, L_2) = (\neg)p(lgg(t_1, s_1), \dots, lgg(t_n, s_n)).$$

The LGG of two clauses $C_1 = \{l_1, \dots, l_k\}$ and $C_2 = \{m_1, \dots, m_n\}$ is defined as:

$$lgg(C_1, C_2) = \{lgg(l_i, m_i) \mid l_i \in C_1 \text{ and } m_i \in C_2 \text{ and } lgg(l_i, m_i) \text{ is defined}\}.$$[2]

As a simple example, consider the clauses

$$C_1 : win(conf1) \leftarrow occ(place1, x, conf1), occ(place2, o, conf1)$$

and

$$C_2 : win(conf2) \leftarrow occ(place1, x, conf2), occ(place2, x, conf2).$$

C_1 and C_2 represent two winning configurations in a two-person game with two places that can be occupied by an "x" or an "o". The lgg of the two clauses is:
$lgg(C_1, C_2) =$
$win(Conf) \leftarrow occ(place1, x, Conf), occ(L, x, Conf), occ(M, Y, Conf), occ(place2, Y, Conf).$
The intuitive meaning of this clause is that a position is winning if it contains an "x" in the first place and something in the second. The LGG of n clauses $lgg(C_1, C_2, \dots, C_n)$ is $lgg(C_1, lgg(C_2, \dots, C_n))$.

2.4.2 Relative Least General Generalization and Golem

Although least general generalization can provide a theoretical basis for generalization in the ILP setting, the simple LGG of a set of clauses alone is far from sufficient

[2] If $lgg(l_j, m_j)$ is not defined for some j, then the pair of terms l_j and m_j are simply ignored.

for the ILP problem. In fact, we are mostly interested to find the generalization of a set of examples in relation to a background knowledge and theory. The least general generalization of two clauses w.r.t. given background knowledge is called *relative* least general generalization (RLGG) (Plotkin, 1971). Note that the only difference between RLGG and LGG is that the former takes the (restricted) set of background knowledge when computing the least general generalization of a pair of examples but the latter does not. All the logical properties of LGG, therefore, carry to RLGG.

The RLGG of two examples can produce a very large clause with a lot of redundant literals in its body. To reduce the clause size, the search only considers a restricted model of the background knowledge K, the h-easy model which is the set of all Herbrand instantiations of h-easy atoms of K. (An atom a is h-easy with respect to K if there is a derivation of a from K involving at most h resolutions.)

Example 1 shows the result of taking the RLGG of a given pair of positive examples from learning the concept '$\mathsf{qsort/2}$' (which sorts a given list) given background predicates like '$\mathsf{append/2}$' (which appends two lists) and '$\mathsf{partition/4}$' (which splits a list into two lists). From this example, one could imagine the size of the resulted clause of taking the RLGG of two examples.

Example 1 The following is the RLGG of C_1 and C_2:

$C_1 = \mathsf{qsort([1],[1])} \leftarrow$

$\mathsf{append([],[1],[1])},$

$\mathsf{append([0,1],[2,3,4],[0,1,2,3,4])}, \ldots,$

$\mathsf{partition(1,[],[],[])},$

partition(2,[4,3,1,0],[1,0],[4,3]),...,

qsort([],[]),

qsort([1,0],[0,1]),

qsort(4,3],[3,4]), ...

C_2 = qsort([2,4,3,1,0],[0,1,2,3,4]) $\leftarrow$

append([],[1],[1]),

append([0,1],[2,3,4],[0,1,2,3,4]),...,

partition(1,[],[],[]),

partition(2,[4,3,1,0],[1,0],[4,3]),...,

qsort([],[]),

qsort([1,0],[0,1]),

qsort(4,3],[3,4]), ...

$C_3 = lgg(C_1, C_2)$

= qsort([A | B], [C | D]) $\leftarrow$

append(E,[A | F],[C | D]),

append([],[1],[1]),

append(G,[H | I],[J | K]),

append([0,1],[2,3,4],[0,1,2,3,4]),

partition(A,B,L,M),

partition(1,[],[],[]),

```
partition(H,N,O,P),
partition(2,[4,3,1,0],[1,0],[4,3]),
qsort(L,E),
qsort([],[]),
qsort(O,G),
qsort([1,0],[0,1]),
qsort(M,F),
qsort(P,I),
qsort([4,3],[3,4]), ...
```

GOLEM (Muggleton & Feng, 1992) also contains an outer loop that finds a set of consistent clauses covering the positive examples like FOIL. However, it builds a clause by considering the *relative least general generalization* (RLGG) of random pairs of positive examples.

GOLEM starts by taking a sampling of RLGGs of pairs of uncovered positive examples and chooses the one that has the greatest coverage for further generalization. It stops building the clause when this RLGG cannot be further generalized (i.e when any further generalization produces inconsistent clauses.) Figure 2.2 is a summary of the algorithm.

2.4.3 Inverse Resolution and CIGOL

Resolution is a sound rule of inference in (mechanical) theorem proving (Robinson, 1983). Given two clauses C_1 and C_2, the resolution rule allows one to deduce a new clause as follows. Assume the two clauses are variable disjoint (i.e. they share no

Procedure *Golem*
Input:
ξ^+: the set of positive examples
ξ^-: the set of negative examples
Output:
H: the set of learned clauses

$H := \emptyset$
$Pairs :=$ random sampling of pairs from ξ^+
$RLggs := \{C : \langle e, e' \rangle \in Pairs$ and $C = RLGG(e, e')$ and C consistent wrt ξ^- $\}$
$S :=$ the pair $\{e, e'\}$ whose RLGG has the greatest cover in $RLggs$
Do
  $Examples :=$ a random sampling of examples from ξ^+
  $RLggs := \{C$: $e' \in Examples$ and $C = RLGG(S \cup \{e'\}))$ and C consistent wrt ξ^- $\}$
  Find e' which produces greatest cover in $RLggs$
  $S := S \cup \{e'\}$
  Add $RLGG(S)$ to H
  $Examples := Examples - cover(RLGG(S))$
While increasing-cover
Return H
End Procedure

Figure 2.2: The GOLEM algorithm

common variables), and let literals l_1 and l_2 belong to C_1 and C_2 respectively. Let θ be the most general unifier (MGU) of l_1 and l_2 such that $\neg l_1\theta = l_2\theta$.

Since l_1 and l_2 are variable disjoint, this can be rewritten as $\neg l_1\theta_1 = l_2\theta_2$, with $\theta = \theta_1\theta_2$. By the resolution rule, the resolvent C of parent clauses C_1 and C_2 is:

$$C = (C_1 - \{l_1\})\theta_1 \cup (C_2 - \{l_2\})\theta_2.^3 \tag{2.5}$$

Resolution is a basis for deductive systems as much as the inversion of resolution (or simply inverse resolution) can be a basis for developing inductive (learning) systems. In other words, if a correct hypothesis, together with background knowl-

[3] This operation is also denoted simply as $C = C_1 \cdot C_2$.

edge, can be used in a resolution proof of some examples, then that hypothesis can be induced from the background knowledge and the examples by inverting the resolution process.

More precisely, inverse resolution concerns computing C_2 given the resolvent C and a parent clause C_1. However, there can be multiple solutions to C_2 (Muggleton & Buntine, 1988). In fact, to find C_2, one needs to decide if and how to turn terms involved in the resolution operation into variables. The ILP system CIGOL (Muggleton & Buntine, 1988) has three (generalization) operators to build clauses from given examples and a background theory, namely absorption (the "V" operator), intra-construction (the "W" operator), and truncation. We will only briefly discuss the absorption operator.

The absorption operator in CIGOL constructs C_2 given C_1 and C. Since $\neg l_1\theta_1 = l_2\theta_2$ (where $\theta = \theta_1\theta_2$), we have

$$l_2 = \neg l_1\theta_1\theta_2^{-1}$$

By manipulating equation 2.5, we obtain a formal definition of the absorption operator:

$$C_2 = (C - (C_1 - \{l_1\})\theta_1)\theta_2^{-1} \cup \{\neg l_1\}\theta_1\theta_2^{-1} \tag{2.6}$$

There are two assumptions in CIGOL: 1) the clauses $(C_1 - \{l_1\})\theta_1$ and $(C_2 - \{l_2\})\theta_2$ must not contain common literals (called the separability assumption) and 2) C_1 must be a unit clause, i.e., $C_1 = \{l_1\}$, which is called the unit clause assumption. These constraints simplify equation 2.6 to:

$$C_2 = (C \cup \{\neg l_1\}\theta_1)\theta_2^{-1} \tag{2.7}$$

To solve equation 2.7, one must compute an inverse substitution θ_2^{-1}. This requires one to decide which terms and subterms within $(C \cup \{\neg l_1\}\theta_1)$ map to distinct variables, which unfortunately leads to combinatorial explosion. A best-first search algorithm (using compaction as the search heuristic) is implemented in CIGOL to compute C_2 given C and l_1 (Muggleton & Buntine, 1988), which will not be discussed here.

Finally, CIGOL is an interactive learning system which incrementally constructs first-order Horn clause theories from example clauses presented by a human teacher. Questions are asked of the teacher to verify generalizations made by the various generalization operators.

2.5 Inverse Entailment and PROGOL

2.5.1 The Theory on Inverse Entailment

Inverse implication (Muggleton, 1992) has been the core problem in Inductive Logic Programming since induction can be treated as the inverse of deduction (Muggleton, 1999). Earlier approaches to the problem involved inverting resolution in theorem proving (Muggleton & Buntine, 1988). However, such approaches are incomplete since inverting θ-subsumption is incomplete (Plotkin, 1970). It has been shown by (Plotkin, 1970) that if C θ-subsumes D, then C implies D (i.e. $C \models D$). However, he also noted that C implies D does not necessarily mean that C θ-subsumes D. That is to say, if we perform generalization under θ-subsumption of a set of clauses S, we can fail to find a suitable generalization, even if there exists a clause C such that $C \models S$.

Eventually, the discovery that such a distinction between θ-subsumption and implication between a pair of clauses C and D is only relevant when C can self-resolve (Muggleton, 1992) has led to the development of the precise conditions under which a clause C implies another clause D. There are different ways of formalizing these conditions and the one adopted in (Muggleton, 1995) has been widely known as *inverse entailment*, as it is grounded in model theory.

Here is the basic idea in inverse entailment. The general problem specification of ILP is that given background knowledge B and examples E find the best (or simplest) consistent hypothesis H such that

$$B \wedge H \models E \tag{2.8}$$

This can be rearranged to

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

Let $\overline{\bot}$ be the conjunction of ground literals which are true in all models of $B \wedge \overline{E}$. (It exists if B and E are definite logic programs.) Since $\overline{H}$ is true in every model of $B \wedge \overline{E}$, therefore we have

$$B \wedge \overline{E} \models \overline{\bot} \models \overline{H}$$

So, for all H

$$H \models \bot. \tag{2.9}$$

Usually, only the case where H and E are single clauses is considered. It is clear that any H that satisfies equation 2.8 also satisfies equation 2.9. Thus, one needs only to find solutions to equation 2.9. The following theorem characterizes conditions under which a solution exists (Muggleton, 1995):

Theorem 1. Let C and D be definite clauses and $S(D)$ be the sub-saturants of D. $C \models D$ iff one of the following conditions hold:

1. D is a tautology,

2. C θ-subsumes D,

3. C θ-subsumes $C' \in S(D)$.

Proof. The details on the proof of this theorem are contained in (Muggleton, 1995) and will be omitted here. $\Box$

The definition of sub-saturants is given in (Muggleton, 1995). Intuitively, $S(D)$ is the set of clauses subsumed by some C such that a Herbrand model of $C \wedge \overline{D}$ does not exist (which is the condition under which $C \models D$). The third condition corresponds to the case when C can self-resolve but as it is remarked in (Muggleton, 1992) that in most real world applications this case is not significant, to find solutions to equation 2.9, one usually just considers the second condition.

The ILP algorithm Progol (Muggleton, 1995) is an implementation of the theory of inverse entailment. Progol searches only the subsumption lattice of the bottom clause, $\perp$, to find solutions to H. This means that Progol only considers hypotheses H such that

$$\Box \preceq H \preceq \perp$$

where $\Box$ is the so called "empty" clause which denotes the empty set.

There are two notes we need to make here. First, a lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound. Since the relation that orders the set (of hypotheses) here is θ-subsumption ($\preceq$). Hence, the term "subsumption lattice" is used. Second, the empty clause and the most general clause are the same thing; the former refers to an empty set of literals, the latter has an empty set of literals in its body. Thus, they will be used interchangeably unless a distinction is necessary. The empty clause is the least upper bound of the subsumption lattice and the bottom clause is the greatest lower bound.

As we shall discuss in Section 2.5.3 to Section 2.5.5, PROGOL searches for a good clause to add to the building theory by first constructing the most specific clause (the bottom clause), a step commonly called *saturation*, and then searches the subsumption lattice (of clauses bounded between the empty clause and the bottom clause) in a FOIL-like manner (i.e., from general to specific). So, PROGOL is an approach that combines top-down and bottom-up approaches.

2.5.2 Mode Declarations

Since the bottom clause can have an infinite number of literals in its body, PROGOL uses a technique called "mode declaration" (which contains a recall bound, and input-output modes of the variables for a predicate) together with the variable depth bound to constrain the size of the bottom clause. Mode declarations are divided into two kinds: head and body where the former is the mode declaration for the target predicate and the latter are mode declarations for all the predicates in the given set of background knowledge provided to the system.

A mode declaration has either the form $modeh(n, atom)$ (the head mode declaration) or $modeb(n, atom)$ (a body mode declaration), where n, the recall bound, is an integer greater than zero or '*' and *atom* is a ground atom. Terms in the atom are either normal or a place-marker. A normal term is either a constant or function symbol followed by a bracketed tuple of terms. A place-marker is either +type, −type, or #type where type is a constant.

The recall bound is the maximum number of alternative solutions for instantiating the atom used by the algorithm. A recall of '*' indicates all solutions. +type, −type, and #type correspond to input variables, output variables, and constants respectively.

A sample set of mode declarations for a grammar learning problem is given below:

```
:- modeh(1,s(+wlist,-wlist)).
:- modeb(1,prep(+wlist,-wlist)).
```

`modeh(1,s(+wlist,-wlist))` is the (head) mode declaration because the target concept to learn is $s(X, Y)$ where the list of words, X union Y, is a sentence accepted by the grammar. In this mode declaration, the first argument of the target predicate is declared to be an input variable of type *wlist* ("Word list") and it has a recall bound of 1. `modeb(1,prep(+wlist,-wlist))` is a (body) mode declaration for a background predicate $prep(X, Y)$ which recognizes if the first word in the list of words X is a preposition and Y is the rest of the list of words in X. Similarly, the first argument is declared to be an input variable of type *wlist*.

The rest of the body mode declarations for the simple grammar learning problem is given below:

```
:- modeb(1,det(+wlist,-wlist)).
:- modeb(1,noun(+wlist,-wlist)).
:- modeb(1,tverb(+wlist,-wlist)).
:- modeb(1,iverb(+wlist,_wlist)).
:- modeb(*,np(+wlist,-wlist)).
:- modeb(*,vp(+wlist,-wlist).
```

where each mode declaration corresponds to a distinct syntactic category.

The pair of lists X and Y in $s(X, Y)$ is also more formally called a *difference list* in the context of parsing. More precisely, a difference list consists of two lists: 1) an ordinary list, and 2) a pointer to the tail of the ordinary list. The intuition is that it represents the ordinary list minus the elements in the tail. In PROLOG notation, the following pairs are all representations of the same list, which has elements a, b, and c:

```
[a,b,c]        []
[a,b,c,d,e]    [d,e]
```

2.5.3 A Trace of the Construction of the Most Specific Clause

We are going to give a trace of the construction of the most specific clause using the simple grammar learning problem mentioned. Suppose a randomly chosen example e to generalize is:

`s([the,man,walks,the,dog],[]).`

From the head mode declaration `modeh(1,s(+wlist,-wlist))` we have the (trivial) deduction:

$$B \wedge \overline{e} \models \overline{s([the, man, walks, the, dog], [])}$$

From the body mode declaration `modeb(1,det(+wlist,-wlist))` and replacing the input variable by [the,man,walks,the,dog] we have the deduction:

$$B \wedge \overline{e} \models det([the, man, walks, the, dog], [man, walks, the, dog])$$

Note that this constructs the term [man,walks,the,dog] in place of the output variable. Using the body mode declaration `modeb(1,noun(+wlist,-wlist))` with this new term and replacing the input variable by the new term we have the deduction:

$$B \wedge \overline{e} \models noun([man, walks, the, dog], [walks, the, dog])$$

However, using other mode declarations in a similar way we can get the following deductions as well:

$$\begin{aligned} B \wedge \overline{e} &\models np([man, walks, the, dog], [walks, the, dog]) \\ B \wedge \overline{e} &\models verb([walks, the, dog], [the, dog]) \\ B \wedge \overline{e} &\models vp([walks, the, dog], [the, dog]) \\ B \wedge \overline{e} &\models np([the, dog], []) \end{aligned}$$

Putting these and all other similar deductions together we get

$$\begin{aligned} B \wedge \overline{e} \models \; & \overline{s([the, man, walks, the, dog], [])} \wedge \\ & det([the, man, walks, the, dog], [man, walks, the, dog]) \wedge \ldots \wedge \\ & np([the, dog], []) \end{aligned}$$

$\overline{\bot}$ is the right hand side of the above deduction. Actually, a restricted minimal Herbrand model has been constructed for $B \wedge \overline{E}$, the mode declarations being used to guide the inclusion of predicates that might be of importance. To derive $\bot$, the above is first negated to give

$$\begin{gathered} s([the, man, walks, the, dog], []) \vee \\ \overline{det([the, man, walks, the, dog], [man, walks, the, dog])} \vee \ldots \vee \\ \overline{np([the, dog], [])} \end{gathered}$$

and then the most specific clause can be constructed by replacing terms in the above by unique variables in a step commonly called anti-instantiation (or anti-unification):

$$\begin{aligned}\perp \quad = \quad & s(A,B) \vee \\ & \overline{det(A,C)} \vee \\ & \overline{np(A,D)} \vee \\ & \overline{noun(C,D)} \vee \\ & \overline{tverb(D,E)} \vee \\ & \overline{iverb(D,E)} \vee \\ & \overline{vp(D,E)} \vee \\ & \overline{det(E,F)} \vee \\ & \overline{np(E,B)}.\end{aligned}$$

Thus, $\perp$ is

$$\begin{aligned}s(A,B) \quad \leftarrow \quad & det(A,C), \\ & np(A,D), \\ & noun(C,D), \\ & tverb(D,E), \\ & iverb(D,E), \\ & vp(D,E), \\ & det(E,F), \\ & np(E,B).\end{aligned}$$

2.5.4 Bottom Clause Construction Algorithm

Figure 2.3 shows the algorithm for constructing the bottom clause in PROGOL. The recall bound is used to determine how many times to call the Prolog interpreter for each instantiation of the clause in step 4. It may well be that the clause will succeed many times and produce many answer substitutions. The maximum variable depth determines how many times step 4 is executed. The Prolog interpreter inside PROGOL is also bounded in its number of resolution steps and in its depth.

Let e be the clause $a \leftarrow b_1, \ldots, b_n$.
Then $\overline{e}$ is $\overline{a} \wedge b_1 \wedge \ldots \wedge b_n$.
$hash : Terms \rightarrow N$ is a function uniquely mapping terms to natural numbers.

1. Add $\overline{e}$ to the background knowledge
2. $InTerms = \emptyset, \perp = \emptyset$
3. Find the first head mode declarations h such that h subsumes a with substitution θ
   For each v/t in θ,
      if v corresponds to a #type, replace v in h by t
      if v corresponds to a +type or -type, replace v in h by v_k
         where v_k is the variable such that $k = hash(t)$
      if v corresponds to a +type, add t to the set $InTerms$.
   Add h to $\perp$.
4. For each body mode declaration b
      For every possible substitution θ of variables corresponding to +type by terms
      in the set $InTerms$
         Repeat recall times
            If Prolog succeeds on goal b with answer substitution θ'
               For each v/t in θ and θ'
                  If v corresponds to #type, replace v in b by t
                  otherwise replace v in b by v_k where $k = hash(t)$
                  If v corresponds to a -type, add t to the set $InTerms$
               Add $\overline{b}$ to $\perp$
5. Increment the variable depth
6. Goto step 4 if the maximum variable depth has not been achieved
7. Return $\perp$.

Figure 2.3: PROGOL's algorithm for constructing the bottom clause.

2.5.5 The Progol Algorithm

Progol uses a simple set covering algorithm like that of Foil in which each iteration performs the following steps: 1) Randomly chooses an example (a.k.a. seed example) from the set of *uncovered* positive examples, 2) Finds a clause (with maximal compression defined below) that generalizes the seed example chosen in step 1), 3) Adds the clause found to the building theory, and 4) Positive examples covered by the clause are removed. These steps repeat, in the order from 1) to 4), until there remains no uncovered positive examples.

In order to find the clause with maximal compression [4], Progol searches the subsumption lattice with an A*-like algorithm. A simple outline of this algorithm is given in Figure 2.4.

Suppose E is the example being generalized.

1. $Open = \{\Box\}$, $Closed = \emptyset$
2. $s = best(Open), Open = Open - \{s\}, Closed = Closed \cup \{s\}$
3. if $prune(s)$ goto 5
4. $Open = (Open \cup refinements(s)) - Closed$
5. if $terminated(Closed, Open)$ return $best(Closed)$
6. if $Open = \emptyset$ return E (no generalization)
7. goto 2

Figure 2.4: Progol's algorithm for searching the subsumption lattice.

The followings are calculated for each candidate clause s: 1) p_s = the number of positive examples covered by s, 2) n_s = number of negative examples covered by s, 3) c_s = length of the clause s - 1, 4) h_s = minimum number of further atoms to

[4] The clause with the "maximal compression" is the one that minimizes the size of the clause and yet maximizes the coverage on the set of positive examples.

complete the clause, 5) $f_s = p_s - (n_s + c_s + h_s)$.

h_s is calculated by inspecting the output variables in the clause and determining whether they have been defined. For example, the clause $s(A, B)$., would have $h_s = 3$ because it requires at least three literals from $\perp$ to construct a chain of atoms connecting A to B. This is found from a static analysis of $\perp$.

f_s is a measure of how well a clause s explains all the examples with preference given to shorter clauses. The function $best(S)$ returns a clause $s \in S$ with the highest f value in S.

$prune(S)$ is true if $n_s = 0$ and $f_s > 0$. In this case, it is not worth considering refinements of s as they cannot possibly do better since any refinement will add another atom to the body of the clause and so cannot have a higher value of p than s does. It also cannot improve upon n_s as the latter is zero. $terminated(S, T)$ is true if $s = best(S), n_s = 0, f_s > 0$ and for each t in T it is the case that $f_s \geq f_t$. In other words none of the remaining clauses nor any potential refinements of them can possibly produce a better outcome than the current one.

This algorithm is guaranteed to terminate and to return the clause (if it exists) which has maximal compression. In the worst case, it will consider all clauses in the subsumption lattice.

2.5.6 Complexity of PROGOL's Bottom Clause

Theorem 2. Let $|\mathcal{M}|$ be the cardinality of $\mathcal{M}$ (the set of mode declarations). Let the number of +type and -type occurrences in each *modeh* be bounded by constants j^- and j^+ respectively. Let the number of +type and -type occurrences in each *modeb* in $\mathcal{M}$ be bounded by constants j^+ and j^- respectively. Let the recall of each mode $m \in \mathcal{M}$ be bounded by the constant r. The cardinality of $\perp_i$ (the bottom clause generated given a variable depth bound i) is bounded by $(r|\mathcal{M}|j^+j^-)^{ij^+}$.

Proof. The details on the proof of this theorem are contained in (Muggleton, 1995) and will be omitted here. □

The theorem shows that the complexity on the size of the bottom clause constructed by PROGOL is exponential w.r.t. the variable depth bound i, which leads to searching a hypothesis space doubly exponential in complexity since the number of hypotheses in the subsumption lattice is two to the power of the size (i.e. no. of literals in the body) of the bottom clause. One can significantly reduce the size of the bottom clause by a new ILP approach we call BETH outlined in Chapter 5.

2.5.7 ALEPH

ALEPH is a publicly available ILP system implemented in a Prolog compiler called Yap (version 4.3.22) that is a generalization of PROGOL. [5] Earlier incarnations (under the name P-Progol) originated in 1993 as part of a funded project undertaken by Ashwin Srinivasan and Rui Camacho at Oxford University. ALEPH can be used to emulate PROGOL. However, it is enriched with more options than PROGOL to configure the search for a good clause. For example, one can put a limit on the amount of CPU time given to theorem proving.

[5]The Aleph Manual can be accessed via
`http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html`.

Chapter 3

Learning Semantic Parsers: Using the CHILL Framework

3.1 Introduction

Earlier approaches to parsing relied on hand-crafted rules developed by experts (e.g. a linguist), which bear the problems mentioned in Chapter 1. The CHILL system (Constructive Heuristics Induction for Language Learning) (Zelle, 1995) represents an approach to *learning* relevant contextual information (tantamount to linguistic knowledge for parsing that could have been extracted from an expert) for the task of disambiguation given complete contexts (i.e. the entire parse state) instead of relying on handcrafting features for parsing. Screenshots of a natural language interface (NLI) we developed with the CHILL system are shown in appendix A.

3.2 Background on Semantic Parsing

The early work on semantic parsing can be dated back to the 70's (Reeker, 1976; Siklossy, 1972) with emphasis on discovering learning mechanism for language acquisition and cognitive modelling of human language learning. While some focused on cognitive modelling of language acquisition, others focused on building realistic NLP applications.

Traditional NLP approaches to tackling tasks like building NLIs for databases include *augmented transition networks* (Woods, 1970) which operationalize context-free grammars for producing semantic representations, *semantic grammars* (Hendrix, Sacerdoti, Sagalowicz, & Slocum, 1978; Brown & Burton, 1975) which are context-free grammars in which non-terminals are used to represent domain specific concepts (instead of syntactic categories), and *logic grammars* (Abramson & Dahl, 1989; Warren & Pereira, 1982) which encode linguistic dependencies and structure building operations using logical unification.

Traditional (*rationalist*) approaches to constructing semantic parsers very often involve hand-crafting of expert knowledge represented as rules (maybe with limited automation). However, hand-crafted parsers suffer from problems with robustness and incompleteness, even for domain specific applications. As the task scales up in size, hand-crafting becomes more and more difficult which is the so-called problem of *knowledge engineering bottleneck* that exists in many interesting AI domains. This results in applications that are time-consuming to build and yet perform poorly – incomplete, inefficient, and brittle.

More recent approaches, therefore, have shifted from this knowledge engineering perspective to a more empirical based paradigm where parsers are constructed through learning algorithms which use a large corpus of training data. For instance, Miller (1995) presents a statistical approach to the task of mapping flight information requests (in English) to SQL which could be used to access the relevant flight information. The frame (or semantic) representation for a given parse tree of the user request which can be further transformed into an SQL is choosen based on statistics collected from training data. A method based on semantic classification trees for parser construction is described in (Kuhn & De Mori, 1995). The classification trees which are used for semantic interpretation are learned from a corpus of training data. Zelle (1995) employs *inductive logic programming* techniques to learn control rules to "specialize" the parser acquired by the CHILL system.

3.3 The CHILL Approach

We are going to provide a brief discussion of the system here to explain the working of the parser and how contextual information can be learned and utilized for the parsing operators. The natural language interface developed for a U.S. Geography database is used as an example application here.[1] Further details on the system can be found in (Zelle, 1995).

The (syntactic) structure of a sentence is not enough to express its meaning. For instance, the NP *the catch* can have different meanings depending on whether one is talking about a baseball game or a fishing expedition. To talk about different possible readings of the phrase *the catch*, one therefore has to define each specific sense of the phrase. The representation of the context-independent meaning of a sentence is called its *logical form* (Allen, 1995).

[1] It is available via `http://www.cs.utexas.edu/users/ml/geo.html`.

Database Category	*Database Objects*
City	cityid(austin,tx)
State	stateid(mississippi)
River	riverid(mississippi)
Place	placeid('death valley')

Table 3.1: Sample of objects and categories in the Geography database

Database items can be ambiguous when the same item is listed under more than one attribute (i.e. a column in a relational database). For example, the term "Mississippi" is ambiguous between being a river name or a state name, in other words, two different *logical forms*, in our U.S. Geography database. The two different senses have to be represented distinctly for an interpretation of a user query. Databases are usually accessed by some well defined structured languages, for instance, SQL. These languages bear certain characteristics similar to that of logic in that they require the expression of quantification of variables (the attributes in a database) and the notion of logical operations[2] on them. The different pieces of information in a database may also be related to each other and this relational knowledge could be useful for constructing the parser. First order logic, therefore, becomes our choice of knowledge representation framework for these logical forms of all the database objects, relations and any other information related to representing the meaning of a user query. However, it is not the case that the parser used in CHILL can only work with a strictly logical representation. The choice of a representational scheme is flexible. For instance, CHILL is also applied to a database containing facts about Northern California restaurants and the semantic representation scheme resembles SQL. Some examples of the semantic representation of database items of

[2]For instance, in SQL, we have AND and OR to express the logical relationships between constraints on the attributes over the query.

Predicates	*Description*
city(C)	C is a city
capital(S,C)	C is the capital of state S
density(S,D)	D is the population density of state S
loc(X,Y)	X is located in Y
len(R,L)	L is the length of river R
next_to(S1,S2)	state S1 borders S2
traverse(R,S)	river R traverses state S

Table 3.2: Sample of predicates of interest for a database access

the U.S. Geography database are shown in Table 3.1.

We will briefly describe the language used for representing the meaning of a natural language query, the parsing framework employed, and the approach that is taken in CHILL for parser acquisition.

3.3.1 Semantic Representation and the Query Language

The most basic constructs of the representation language are the terms used to describe objects in the database and the basic relations between them. Some examples of objects of interest in the domain are states, cities, rivers and places. We have given semantic categories to these objects. For instance, stateid(texas) represents the database item texas as an object of the database category state. Of course, a database item can be a member of multiple categories.

Database objects do bear relationships to each other or can be related to other objects of interest to a user who is requesting information from it. In fact, a very large part of accessing database information is to sort through tuples that satisfy the constraints imposed by these relationships of database objects in a user query. For instance, in a user query like "What is the capital of Texas?", the data of interest is a city that bears a certain relationship to a state called Texas, or more

Meta-predicates	*Description*
answer(A,Goal)	A is the answer to retrieve in Goal
largest(X, Goal)	X is the largest object satisfying Goal
smallest(X, Goal)	similar to largest
highest(X,Goal)	X is the highest place satisfying Goal
lowest(X,Goal)	similar to highest
longest(X,Goal)	X is the longest river satisfying Goal
shortest(X,Goal)	similar to longest
count(X,Goal,N)	N is the total number of X satisfying Goal
most(X,C,Goal)	X satisfies Goal and maximizes C
fewest(X,C,Goal)	similar to most

Table 3.3: Sample of meta-predicates used in database queries

precisely its capital. The capital/2 relation (or predicate) is ,therefore, defined to handle questions that require them. More of these relations of possible interest to the domain are shown in Table 3.2.

We also need to handle object modifiers in a user query such as "What is the largest city in California?". The object of interest X which belongs to the database category city has to be the largest one in California and it would be represented as largest(X, (city(X), loc(X,stateid(california)))). The meaning of an object modifier depends on the type of its argument. In this case, it means the city X in California that has the largest population (in the number of citizens). To allow predicates to describe other predicates would be a natural extension to the first order framework in handling these kind of cases. These "meta-predicates" have the property that at least one of their arguments take a conjunction of predicates. Finally, an object which is an argument of a certain predicate can appear at a later point in a sentence and this requires the use of a predicate like const(X,Y) (which means the object X equals the object Y) for the parser to work. The use of const/2 will be further explained in the following section where the working of the parser is discussed. A list of meta-predicates is shown in Table 3.3. Some sample database queries for the U.S. Geography domain are shown in Table 3.4.

U.S. Geography
What is the capital of the state with the largest population? answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).
What state has the most rivers running through it? answer(S, most(S, R, (state(S), river(R), traverse(R,S)))).
How many people live in Iowa? answer(P, (population(S,P), const(S,stateid(iowa)))).

Table 3.4: Sample of Geography questions in different domains

3.3.2 Actions of the Parser

The parser presented here that builds a logical query given a sentence is based on a standard *shift-reduce* parsing framework. (A more thorough discussion on shift-reduce parsing can be found in (Allen, 1995; Tomita, 1986).) There is no explicit semantic grammar but the parsing actions are derived from the examples (which is a pair of sentence and its logical query) and they are guaranteed *complete* with respect to them (i.e. there exists a sequence of parsing actions (a derivation) that leads to the right logical query for each sentence). The parser actions are generated from templates given a logical query; an action template will be instantiated to form a specific parsing action. The templates are INTRODUCE, COREF_VARS, DROP_CONJ, LIFT_CONJ, and SHIFT. INTRODUCE pushes a logical form onto the parse stack based on information in the lexicon. COREF_VARS binds two arguments of two different logical forms to the same variable. DROP_CONJ (or LIFT_CONJ) takes a logical form on the parse stack and puts it into one of the arguments of a meta-predicate. DROP_CONJ assumes the logical form precedes the meta-predicate on the parse stack while LIFT_CONJ assumes it is the other way around. SHIFT pushes a word from the input buffer onto the parse stack. Their actions are summarized in Table 3.5. The parsing actions are tried in exactly that order; the set of parsing actions resemble a decision list in which the first applicable choice is taken.

The parser also requires a lexicon to interpret meaning of phrases into specific logical forms. The lexicon can be learned from a given set of sample sentence and query pairs (Thompson & Mooney, 1999). We will briefly illustrate what action each template does here by showing a trace of parsing a simple example:

Sentence:	What is the capital of Texas?
Logical Query:	answer(C, (capital(C,S), const(S, stateid(texas)))).

The first thing we need is a lexicon. A very simple lexicon that maps 'capital' to 'capital(_,_)' and 'Texas' to 'const(_,stateid(texas))' would suffice here. The parser begins with an initial stack and a buffer holding the input sentence which is the initial parse state. Each predicate on the parse stack has an attached buffer to hold the context in which it was introduced; words from the input sentence are shifted onto the (stack) buffer during parsing. The contextual information may be useful for the learning of contextual knowledge for disambiguation. The initial parse state is shown below:

Parse Stack:	[answer(_,_):[]]
Input Buffer:	[what,is,the,capital,of,texas,?]

Since the first three words in the input buffer do not map to any logical forms, the next sequence of steps are three SHIFT actions which result in the following parse state:

Parse Stack: [answer(_,_):[the,is,what]]

Input Buffer: [capital,of,texas,?]

Now, 'capital' is at the head of the input buffer and is mapped to 'capital(_,_)' in our lexicon. The next action to apply is, therefore, INTRODUCE which is actually instantiated to introduce(capital(_,_), [capital], S0, S1). Notice that a particular phrase in general can be mapped to different logical forms due to lexical ambiguities. The contextual knowledge required for the proper interpretation of a phrase is learned by the induction algorithm. The resulting parse state is shown below:

Parse Stack: [capital(_,_):[], answer(_,_):[the,is,what]]

Input Buffer: [capital,of,texas,?]

The next action is a COREF_VARS. We have two possible choices here: coref_vars(capital, 2, 1, answer, 2, 1, S0, S1) or coref_vars(capital, 2, 2, answer, 2, 1, S0, S1). A choice like coref_vars(capital, 2, 1, answer, 2, 2, S0, S1) is eliminated by inspecting if one of the predicates is a meta-predicate and which argument positions hold variables. Since the question is asking about the capital, the first one is the proper choice and we will pick it here. In general, the knowledge required for properly selecting a COREF_VARS action is learned. The resulting parse state is shown below:

Parser actions	Parser action descriptions
INTRODUCE(TERM,PHRASE,SO,S1)	Put TERM on the parse stack of input parse state S0 if PHRASE occurs at the beginning of the input buffer of S0 to produce S1
COREF_VARS(T1,A1,N1,T2,A2,N2,S0,S1)	Unify the N1-th argument of the term T1 with the N2-th argument of the term T2 if T1 and T2 are on the parse stack of S0 having arity A1 and A2 respectively
DROP_CONJ(T1,AR1,T2,AR2,N2,S0,S1)	Place the term T1 in the N2-th argument of the term T2 to form a new conjunct if T1 comes before T2 on the parse stack of S0 having arity AR1 and AR2 respectively
LIFT_CONJ(T1,AR1,T2,AR2,N2,S0,S1)	Similar to DROP_CONJ except that the term T2 comes before the term T1 on the parse stack of S0
SHIFT(S0,S1)	A word at the beginning of the input buffer of S0 will be shifted into the buffer of the top predicate on the parse stack of S0 if the input buffer is not empty

Table 3.5: A summary of the parser actions

Parse Stack: [capital(C,_):[], answer(C,_):[the,is,what]]

Input Buffer: [capital,of,texas,?]

The next sequence of steps are two SHIFT's followed by an INTRODUCE which is instantiated to introduce(const(_,stateid(texas)), [texas], S0, S1). The resulting parse state is:

Parse Stack: [const(_,stateid(texas)):[], capital(C,_):[of,capital], answer(C,_):[the,is,what]]

Input Buffer: [texas,?]

Notice that instead of looking ahead into the input buffer for 'Texas' and introducing 'capital(_,stateid(texas))', we introduced 'capital(_,_)' and its second argument is left to be instantiated by a COREF_VARS when the parser comes to the term 'Texas'. This helps avoid the problem of having to combine different disambiguation decisions at the same point. For instance, if the question was "What is capital of the state that borders Texas?", we would have to make a decision between introducing 'capital(C,stateid(texas))' or 'capital(C,_)' precisely at the point where 'capital' was at the beginning of the input buffer. It would be easier for the parser to make such decisions when the relevant context become available on the parse stack at a later point.

The next sequence of actions are COREF_VARS which is instantiated to coref_vars(const, 2, 1, capital, 2, 2, S0, S1) and two more SHIFT operations. Again, we have two possible COREF_VARS instantiations here, the proper one was chosen. The resulted parse state is shown below:

Parse Stack:	[const(S,stateid(texas)):[?,texas], capital(C,S):[of,capital], answer(C,_):[the,is,what]]
Input Buffer:	[]

Now, the next steps would be two DROP_CONJ. They are drop_conj(const, 2, answer, 2, 2, S0, S1) and drop_conj(capital, 2, answer, 2, 2, S0, S1). The resulted parse state is:

Parse Stack:	[answer(C, (capital(C,S), const(S,stateid(texas)))):[?,texas,of,capital,the,is,what]]
Input Buffer:	[]

We have reached the final parse state at this point since none of the parser actions can be applied. The logical query constructed is then read off from the parse stack.

3.3.3 Components of the CHILL Architecture

The CHILL (Constructive Heuristic Induction for Language Learning) framework is based on an empirical approach to parser construction integrated in a symbolic knowledge acquisition framework for both the learning and the representation of semantic knowledge. Since the parser is to construct logical queries from natural language input, it would be natural to implement the parser as a logic program where the parsing operators are actually Horn clauses.

Given a corpus of sentence and query pairs, the task is to induce a semantic parser. Inducing a parser directly from these pairs is not feasible since the space of possible parsers would be too large. However, if we begin with an initial parser generated by instantiating the action templates given the examples (an overly general initial theory), the problem could be reduced to learning control rules for it. ILP techniques for learning search control knowledge will be used since a (first-order) logical knowledge representation framework is employed. The idea of learning control rules for a parser can also be traced back to earlier work in acquiring syntactic knowledge for parsing (Berwick, 1985). Figure 3.1 shows the architecture for CHILL.

The working of CHILL is divided into four phases as indicated in the figure: 1) generating the initial parser: using the templates of parsing actions to generate an overly general initial parser complete w.r.t. parsing the set of training examples, 2) analysing the examples: using the overly general parser to parse the training examples and collect positive and negative examples (all spurious parse states are

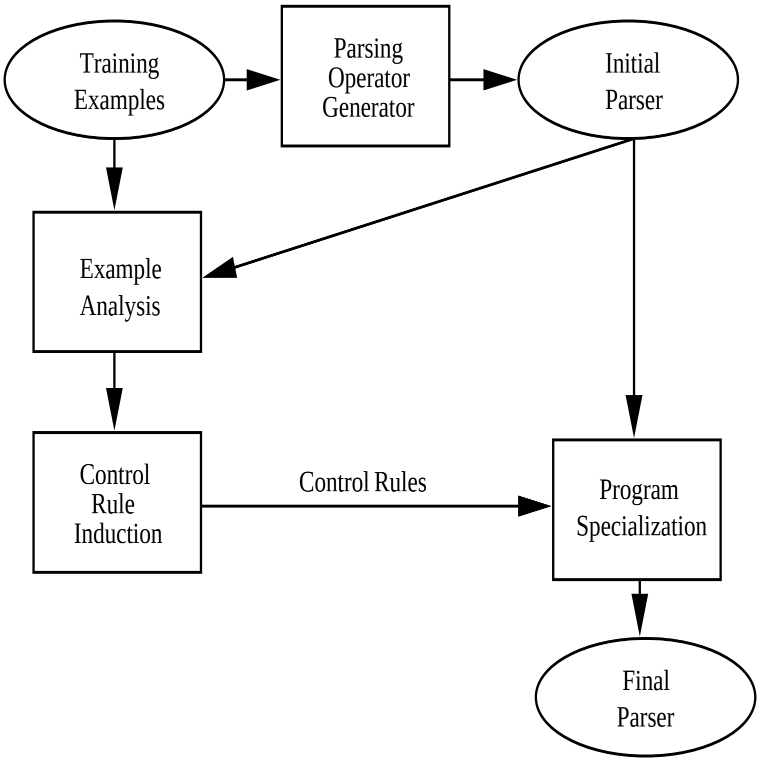


Figure 3.1: The architecture for CHILL

negative examples; otherwise, positive examples of a parsing operator), 3) inducing the control rules: using ILP algorithms to learn a theory for each parsing operator given positive and negative examples for them, and 4) specializing the initial parser: incorporating a learned theory from step 3) as a guard to a parsing operator. We will briefly describe each of them here.

3.3.4 A Minor Improvement to a CHILL's Parsing Operator

We will discuss a minor improvement we made to the COREF_VARS parsing operator which is, perhaps, the most frequently used parsing operator in the CHILL system. In the original CHILL framework, the COREF_VARS parsing operator does not do any "type checking" when binding two arguments of two different terms on the parse stack. More precisely, COREF_VARS(T1, AR1, N1, T2, AR2, N2, S0, S1) unifies the N1-th argument of the term T1 with the N2-th argument of the term T2 if T1 and T2 are on the parse stack of S0 having arity AR1 and AR2 respectively *regardless* of the types of the arguments T1 and T2.

This is normally not a problem except if one of the terms is a constant (i.e. a term of the form `const(X,Y)` where X is an unbound variable and Y is a database object like `stateid(texas)`). Suppose we have the parsing operator `coref_vars(const, 2, 1, river, 1, 1, S0, S1)` which binds the first argument of the term `const/2` with the first and only argument of the term `river/1` on the parse stack of S0. If the parse stack of S0 has two database objects, which can happen if the sentence contains two words referring two different database objects (e.g. "Does the *Rio Grande* river traverse *Texas*?"), this COREF_VARS parsing operator will take the *first* database object appeared on the parse stack and bind it with the term `river/1` regardless of whether it is `const(_,riverid('rio grande'))` or `const(_,stateid(texas))` (the two database objects that can appear on the parse stack of S0).

To solve this problem of potentially binding the wrong object with a term on the parse stack, one can add two additional parameters to the COREF_VARS operator X1 and X2 such that X1 specifies the type of the term T1 and likewise for X2. Thus, our new COREF_VARS operator is defined as follows: COREF_VARS(T1, X1, AR1, N1, T2, X2, AR2, N2, S0, S1) unifies the N1-th argument of the term T1 with the N2-th argument of the term T2 if T1 and T2 are on the parse stack of S0 having arity AR1 and AR2 respectively *and, furthermore, if the type specifier X1 equals X2*. This slightly upgraded version of CHILL is the one employed in Chapter 4.

3.4 CHILLIN

Top-down or bottom-up approaches to ILP have their own strength and weaknesses. For instance, GOLEM requires the use of extensional background knowledge and could result in building clauses with a lot of redundant literals. Specific constraints like using only the h-easy model of the background knowledge have to be enforced to deal with some of these efficiency problems. However, FOIL requires the target hypothesis to be function-free and needs specific constructor functions as part of the background knowledge. For instance, to learn the concept member(X, Y) where X is an element of the (non-empty) list Y. One needs to provide FOIL with a background predicate like head(Y, H) which is like a function that returns the first element H of the list Y. The size of the background knowledge grows with the number of such constructor functions used by the learner, which makes a larger hypothesis space to search. A combination of the two different methods which takes advantage of the strength of each approach may therefore make new ILP approaches better than either alone. CHILLIN[3] was an attempt at combining both approaches and it was used in CHILL for learning control rules. Figure 3.2 shows the outline of CHILLIN.

Unlike set-covering alogrithms like FOIL, CHILLIN consists of a compaction outer loop that builds a more general hypothesis with each iteration. Each iteration finds a clause maximizing the coverage of the set of positive examples (i.e. most

[3] CHILLIN stands for *the CHILL INduction algorithm*.

Procedure *Chillin*
Input:
ξ^+: the $\oplus$ examples
ξ^-: the $\ominus$ example
Output:
DEF: the set of learned clauses

$DEF := \{E \leftarrow \mid E \in \xi^+\}$
Repeat
  $PAIRS :=$ a sampling of pairs of clauses from DEF
  $GENS := \{G \mid G = Find\_A\_Clause(C_i, C_j, DEF, \xi^+, \xi^-)$ for $\langle C_i, C_j \rangle \in PAIRS\}$
  $G :=$ Clauses in $GENS$ yielding most compaction
  $DEF := (DEF -$ (Clauses subsumed by $G)) \cup G$
Until no-further-compaction
Return DEF
End Procedure

Figure 3.2: Outline of the CHILLIN algorithm

compaction). A clause is built by finding the LGG of a random pairs of *clauses* in the building definition *DEF* and if the LGG is overly general, it will be specialized by adding literals to its body like FOIL. The search for a hypothesis is done in a bottom-up manner since it begins with the most specific hypothesis (i.e. the set of positive examples) and continues to generalize it through the compaction loop. The specialization of a clause resembles that of a top-down algorithm as literals are added to its body for specialization and therefore heuristics like information gain can be used to discriminate between literals. Once a clause is found, it will be incorporated into the current theory. Any clause covered (subsumed) by it will be removed from the theory. A novel kind of subsumption is adopted here which is called *empirical subsumption*. A clause $C \preceq_e$ (empirically subsumes) D if the set of ground unit instances covered by the clause C is a subset of that of the clause D.

3.5 Experimental Evaluation

CHILL was applied on a U.S. Geography database and its performance was compared to a hand-crafted semantic parser that came with the database called GEOBASE. A corpus consisting of 250 sentences was built by collecting questions from undergraduate students in the department. The corpus was split into training sets of 225 examples with the remaining 25 held-out for testing. A query is considered correct only if it produces the exact same answer as that of the correct query associated with the test sentence. The recall of the parser is defined as the no. of correct queries produced divided by the no. of test sentences. The number of correct queries produced are the total number of such queries produced in parsing all the test sentences. Appendix E shows the recall (i.e. accuracy) of CHILL's parsers over a 10 trial average. The line labeled "Geobase" shows the average accuracy of the Geobase system.

Chapter 4

Learning Semantic Parsers: Using COCKTAIL

4.1 Introduction

One problem with the CHILLIN induction algorithm is that some contextual information can be lost in the process of performing LGG on a random pair of positive examples, which is the first step in learning a clause. Let's go through a concrete example to illustrate this point. For example, the parser is learning to parse these two sentences: 1) "What is the biggest Texas city?" [1], and 2) "What is the biggest city in Texas?". Suppose we have two intermediate parse states S_a and S_b at which point the parser is about to introduce the predicate `largest(_,_)` on the parse stack given that their input buffers (which contains the sentence or the rest of the sentence to be parsed) are `[biggest,texas,city,?]` and `[biggest,city,in,texas,?]` re-

[1] While this question might sound odd, some user input might not even be completely gramatical. Being able to handle such sentences is, perhaps, the essence of robust parsing.

spectively. The word 'biggest' can be mapped to the predicate `longest(_,_)` in a different context. For example, if the sentence was "What is the biggest *river* in Texas?" The word 'biggest' would have been mapped to the predicate `longest(_,_)` instead. Suppose S_a and S_b are two positive examples of the parsing operator that introduces the predicate `largest(_,_)` on the parse stack given that the first word in the input buffer is 'biggest' and we have a number of negative examples in which 'biggest' was mapped to `longest(_,_)` for the reason mentioned. Now, the LGG of the two input buffers `[biggest,texas,city,?]` and `[biggest,city,in,texas,?]` is the term `[biggest,_,_,_|_]`. The word 'city', which provides important contextual clue for disambiguation (between introducing `largest(_,_)` and `longest(_,_)`), is lost in computing the LGG of the two parse states S_a and S_b.

Obviously, the LGG of two random examples can still make a clause that covers negative examples. However, losing such essential contextual information results in CHILLIN having to rely on inventing predicates to capture these constants in a parse state to specialize the current clause, which very often led to memorizing specific constants and, thus, learning theories that tend to overfit the data. Even worse, sometimes, there is only one positive example (state) available in learning. If such a scenario arises, the parser will just memorize the specific positive example since performing LGG requires having at least a pair of examples.

Paradoxically, sometimes "losing" these constants lead to better generalization. This can happen when *only* the *structure* of a parse state matters. The structure of a parse state is actually itself information useful for resolving ambiguities. For example, the parser is learning to parse these two sentences: 1) "What state has the highest population density?", and 2) "Which city has the highest population?". Suppose we have two intermediate parse states S_1 and S_2 for sentences one and two respectively such that $S_1 =$ `ps([state(_):[the,has,state],answer(_,_):[what]], [highest,population,density,?])`, and $S_2 =$ `ps([city(_):[the,has,city], answer(_,_):[which]],[highest,population,?])`. These two parse states represent the point at which the parser is about to introduce the predicate `largest(_,_)` on the parse stack given that the first word 'highest' in the input buffers can be mapped to this predicate (in the lexicon). However, the word 'highest' can also be mapped to the predicate `highest(_,_)` in a different context. For example, if the parser was learning the sentence "What state has the highest *mountain* in the U.S.?" The word 'highest' would have been mapped to the predicate `highest(_,_)`. Suppose S_1 and S_2 are two positive examples of the parsing operator that introduces the predicate `largest(_,_)` on the parse stack (given that 'highest' is the first word in the input buffer). The LGG of S_1 and S_2 is:

`ps([_:[the,has,_],answer(_,_):[_]],[highest,population,_|_]),`

which captures the relevant structure of the parse state by specifying that 1) the first predicate on the parse stack can be anything, _, and its associated context has to be of the form `[the,has,_]` (i.e. consisting of exactly three words in which the first two are 'the' and 'has'), 2) the input buffer has *at least* three words which starts with the phrase 'highest population' and so on.

Alternatively, one could explicitly mention that the predicate `largest(_,_)` can be introduced on the parse stack given that the first word in the input buffer is 'highest' if 1) the first predicate on the parse stack is `state(_)`, the word 'state' is part of its associated context, and the word 'population' is in the input buffer, *or* 2) `city(_)` is the first predicate on the parse stack, 'city' is part of its assoicated context, and 'population' is in the input buffer by using two clauses (each describing a point). The problem with using background knowledge to explicitly mention the same conditions, in this case, is that the size of the hypothesis (two clauses) is bigger than a simple LGG. In general, the LGG of the positive examples, which captures all the necessary contextual information for disambiguation (like in the above example), gives a smaller (thus more compressive) hypothesis than one using background knowledge that explicitly describes the necessary contextual information. Therefore, the clause constructed from the LGG is more likely going to give a better generalization. For example, hypothetically say if 'town' was another database object which is another geographic unit smaller than a city, then the above LGG would allow the introduce operator to correctly generalize to a new sentence like "What town has the highest population?" while the hypothesis (consisting of the two specific clauses mentioned) would not. Both types of clauses are, hence, necessary for learning.

One can take care of both needs - the need for learning relevant contextual information before they are destroyed in the process of computing LGG and the need for learning specific structures of parse states - by separating the learning of each explicitly. Traditional top-down approaches given relevant background knowledge (e.g, the presence or absence of a certain word or phrase in the input buffer of a parse state) are more suitable for learning contextual information for disambiguation. Bottom-up approaches using LGG for generalization like GOLEM is stronger in terms of learning structures of parse states. By combining both ILP approaches, one can learn both types of parsing features in one coherent mechanism. We will discuss an implementation of this approach we call COCKTAIL (Tang & Mooney, 2001) in Section 4.2.

4.2 Combining Top-down and Bottom-up Approaches in COCKTAIL

A typical ILP algorithm can be viewed as a loop in which a certain *clause constructor* is embedded. A clause constructor is formally defined here as a function $f : T \times B \times E \rightarrow S$ such that given the current theory T, a set of training examples E, and the set of background knowledge B, it produces a set of clauses S. For example, to construct a clause using FOIL (Quinlan, 1990) given an existing partial theory T_p (which is initially empty) and a set of training examples $\xi^+ \cup \xi^-$ (positive and negative), one uses all the positive examples not covered by T_p to learn a single clause C. So, we have $f_{Foil}(T_p, B, \xi^+ \cup \xi^-) = f_{Foil}(T_p, B, \{e \in \xi^+ \mid T_p \not\models e\} \cup \xi^-) = \{C\}$. Notice that f_{Foil} always produces a singleton set. Since different constructors create clauses of different characteristics (like syntax and accuracy), a learner using multiple clause constructors could exploit the various language biases available to produce more expressive hypotheses.

4.2.1 The Clause Constructor of CHILLIN

The details on the ILP algorithm CHILLIN is already given in Section 3.4 and will be omitted here. Let's define the clause constructor for CHILLIN $f_{Chillin}$. Given a current partial theory T_p (initially empty), background knowledge B, and $\xi^+ \cup \xi^-$ as inputs, $f_{Chillin}(T_p, B, \xi^+ \cup \xi^-) = \{G\}$ where G is the clause with the best coverage learned by going through the compaction loop for one step. Although CHILLIN only learns the clause with the best coverage and adds it to the theory, we will allow $f_{Chillin}$ to return the best n clauses (constructed from the LGG of different random pairs of positive examples) ordered by their coverages on the set of positive examples instead.

4.2.2 The Clause Constructor of mFOIL

The details of the ILP algorithm mFOIL is already given in Section 2.3.1 and will be omitted here. Let's define the clause constructor function f_{mFoil} for mFOIL. Given the current building theory T_p, background knowledge B, training examples $\xi^+ \cup \xi^-$, $f_{mFoil}(T_p, B, \xi^+ \cup \xi^-) = \{C\}$ where C is the most significant clause found in the search beam. Again, we will use a modified version of f_{mFoil} which returns the entire beam of promising clauses when none of them can be significantly refined.

4.2.3 Background Knowledge Used in f_{mFoil}

Contextual information for disambiguating different possible parses of a sentence can be represented as "theory constants" to the inductive learner. For example, in our previous sample trace of parsing the sentence "What is the capital of Texas?" in Section 3.3.2, if the phrase "capital" was mapped to money(M,G) (the amount of money M the state government G has) in the lexicon as well, we would need to disambiguate between introducing capital/2 or money/2 on the parse stack. In this case, the context that is helpful for disambiguating between the two cases is the *absence* of the word government in the input buffer. For example, if the sentence was "What fraction of the Texas government state capital is spent on highway construction?", the word 'capital' would have been mapped to `money(M,G)` instead. If we have the predicate phrase_in_buffer(P,S) (the phrase P, a theory constant, appears in the input buffer of the parse state S) in the background knowledge of the learner, the literal not phrase_in_buffer([government],S) is useful for constructing a control rule for the parsing action introduce(capital(_,_), [capital], S0, S1). Besides phrase_in_buffer(P,S), we also have other background predicates like predicate_on_stack(F/A,S) which is true if there is a predicate with the predicate name F and arity A, F and A are theory constants, that appears on the parse stack of the parse state S. An example of such a predicate is predicate_on_stack(capital/2,S) which checks if the predicate capital(_,_) appears on the parse stack of S. We also have another background predicate phrase_on_stack(P,S) which is true if the phrase P, a theory constant, appears in the context of a predicate on the parse stack of the parse state S. An example is phrase_on_stack([capital],S) which checks if the phrase "capital" appears in the context of a predicate on the parse stack of the parse state S. These are all the background predicates we used in learning.

ILP systems like FFOIL (Quinlan, 1996) do make use of background knowledge that can handle theory constants (e.g. checking if the value of a variable equals zero). However, it requires *a prior* knowledge of the set of constants that will be relevant or necessary for the learning task. This may be possible for domains like learning functional definitions where it would be relatively easier to identify a set of "important" constants that may be relevant to a number of learning tasks (like 0 or 1). In other domains like language learning, however, identifying a set of useful constants that is reasonably comprehensive would be rather difficult as one would be required to have enough prior knowledge of the *relevant* contextual information but this is what the learning system is suppose to find out. Handcrafting some possibilities or throwing in an entire dictionary would be either too ineffective or inefficient. Therefore, instead of engineering them as prior knowledge to the system, we obtain possible theory constants from the training data. This, however, requires the system to generate or extract theory constants from examples given a set of background predicates.

More precisely, the idea is that given a set of positive and negative examples of the target concept and a set of background knowledge, we generate a new set of literals using these background predicates which use constants that appear in the set of examples for the learner. Using phrase_in_buffer(P,S) as an example where the sentence "What is the capital of Texas?" is a positive example and "What fraction of the Texas government state capital is spent on highway construction?" is a negative example for the parsing operator introduce(capital(_,_), [capital], S0, S1), one will generate the following set of literals with theory constants if only one-word phrases are considered:

phrase_in_buffer([what],S)

...

phrase_in_buffer([texas],S)

phrase_in_buffer([?],S)

not phrase_in_buffer([what],S)

...

not phrase_in_buffer([texas],S)

not phrase_in_buffer([government],S)

...

not phrase_in_buffer([spent],S)

```
not phrase_in_buffer([on],S)
not phrase_in_buffer([highway],S)
not phrase_in_buffer([construction],S)
not phrase_in_buffer([?],S)
```

4.2.4 The COCKTAIL Algorithm

A set of clause constructors (like FOIL's or GOLEM's) have to be chosen in advance. The decision of what constitutes a sufficiently rich set of constructors depends on the application one needs to build. Although an arbitrary number of clause constructors is permitted (in principle), in practice one should use only a handful of useful constructors to reduce the complexity of the search as much as possible. We have chosen mFOIL's and CHILLIN's clause constructors primarily because the former is the state-of-the-art top-down ILP algorithm while the latter was the best ILP algorithm applied to the problem of learning semantic parsers.

The search of the hypothesis space starts with the empty theory. At each step, a set of potential clauses is produced by collecting all the clauses constructed using the different clause constructors available. Each clause found is then used to *compact* the current building theory to produce a set of new theories; existing clauses in the theory that are empirically subsumed by the new clause are removed. The best one is then chosen according to a given theory evaluation metric and the search stops when the metric score does not improve. The algorithm is outlined in Figure 4.1.

As the "ideal" solution to an induction problem is the hypothesis that has the minimum size and the most predictive power, some form of bias leading the

PROCEDURE COCKTAIL
INPUT:
ξ^+, ξ^-: the $\oplus$ and $\ominus$ examples respectively
f_{mFoil}: the clause constructor function for mFOIL
$f_{Chillin}$: the clause constructor function for CHILLIN
B_{mFoil}: a set of background knowledge for f_{mFoil}
$B_{Chillin}$: a set of background knowledge for $f_{Chillin}$
M: the metric for evaluating a theory
OUTPUT:
T: the learned theory

$T := \{\}$
REPEAT
  $Clauses := f_{mFoil}(T, B_{mFoil}, \xi^+ \cup \xi^-) \cup f_{Chillin}(T, B_{Chillin}, \xi^+ \cup \xi^-)$
  Choose $C \in Clauses$ such that $M((T - \{D : C \preceq_e D\}) \cup \{C\}, \xi^+ \cup \xi^-)$
  is the best
  $T := (T - \{D : C \preceq_e D\}) \cup \{C\}$
UNTIL $M(T, \xi^+ \cup \xi^-)$ does not improve
RETURN T
END PROCEDURE

Figure 4.1: Outline of the COCKTAIL Algorithm

search to discover such hypotheses is desirable. It has been formulated in the *Minimum Description Length (MDL) principle* (Rissanen, 1978) that the most probable hypothesis H given the evidence (training data) D is the one that minimizes the complexity of H given D which is defined as

$$K(H \mid D) = K(H) + K(D \mid H) - K(D) + c \tag{4.1}$$

where $K(\cdot)$ is the Kolmogorov complexity function (Kolmogorov, 1965) [2] and c is a

[2]This function returns the length of the smallest program that computes a given string. It is not Turing-computable because of the halting problem. In practice, one gives an approximation to this function by using a possibly ad hoc scheme of measuring the complexity of an object.

constant. This is also called the *ideal* form of the MDL principle. In practice, one would instead find an H of some set of hypotheses that minimizes $L(H) + L(D \mid H)$ where $L(x) = -\log_2 Pr(x)$ and interpret $L(x)$ as the corresponding Shannon-Fano (or Huffman) codeword length of x. However, if one is concerned with just the *ordering* of hypotheses but not *coding* or *decoding* them, it seems reasonable to use a metric that gives a rough estimate instead of computing the complexity directly using the encoding itself as it would be computationally more efficient.

Now, let $S(H \mid D)$ be our estimation of the complexity of H given D which is defined as

$$S(H \mid D) = S(H) + S(D \mid H) - S(D) \tag{4.2}$$

where $S(H)$ is the estimated prior complexity of H and

$$S(D \mid H) = S(H_1 \cup \{T \leftarrow T', not\ T''\} \cup H' \cup H_2) \tag{4.3}$$

is roughly a worst case complexity of a program (complete and consistent w.r.t D) that results from turning H into a program that knocks out all the negative examples covered by it, and memorizes each individual positive example not covered by it, and includes all the renamed clauses in H (which will be further explained below). Suppose T is the target concept $t(R_1, \cdots, R_k)$ that we need to learn, $T' = t'(R_1, \cdots, R_k)$, and $T'' = t''(R_1, \cdots, R_k)$ are the renaming of the target concept. The right side of Equation 4.3 represents such a complete and consistent program "made from H". It has four components: 1) H_1, a theory that memorizes the set of uncovered positive examples of H, is a set of unit clauses of the form $t(a_1, \cdots, a_k)$ and we have one such unit clause for each positive example not covered by H, and 2) the clause $T \leftarrow T', not\ T''$ (i.e. $t(R_1, \cdots, R_k) \leftarrow t'(R_1, \cdots, R_k), not\ t''(R_1, \cdots, R_k)$), and 3) H' which is exactly the same as H except that any predicate t/k appearing in any clause in H is renamed to t'/k, and 4) H_2, a theory that memorizes each negative example covered by H, is a set of unit clauses of the form $t''(a_1, \cdots, a_k)$ and we have one such unit clause for each tuple $t(a_1, \cdots, a_k)$ in the set of negative examples covered by H. Finding compressive hypotheses that generalize clauses in H_1 and H_2 could be problematic. Thus, we simply take the worst case assuming the discrepancy between H and D is not compressible. A very simple measure is employed here as our complexity estimate (Muggleton & Buntine, 1988). The size S of a set of *Clauses* (or a hypothesis) where each clause C with a *Head* and a *Body* is defined as follows:

$$S(Clauses) = \sum_{C \in Clauses} 1 + termsize(Head) + termsize(Body) \tag{4.4}$$

where

$$termsize(T) = \begin{cases} 1 & \text{if } T \text{ is a variable} \\ 2 & \text{if } T \text{ is a constant} \\ 2 + \sum_{i=1}^{arity(T)} termsize(arg_i(T)) & otherwise. \end{cases} \tag{4.5}$$

The size of a hypothesis can be viewed as a sum of the average number of bits required to encode a symbol appearing in it which can be a variable, a constant, a function symbol, or a predicate symbol, plus one bit of encoding each clause terminator. (Note that this particular scheme gives less weight to variable encoding.) Finally, our theory evaluation metric is defined as

$$M(H, D) = S(H) + S(D \mid H). \tag{4.6}$$

The goal of the search is to find the H that minimizes the metric M. The metric is purely syntactic; it does not take into account the complexity of proving an instance (Muggleton, Srinivasan, & Bain, 1992). However, we are relying on the assumption that syntactic complexity implies computational complexity although this and the reverse are not true in general. So, the current metric does not gaurantee finding the hypothesis with the shortest proof of the instances.

4.3 Experiments

4.3.1 Domains

Two different domains are used for experimentation here. The first one is the United States Geography domain. The database contains about 800 facts implemented in Prolog as relational tables containing basic information about the U.S. states like population, area, capital city, neighboring states, and so on. The second domain consists of a set of 1000 computer-related job postings, such as job announcements, from the USENET newsgroup `austin.jobs`. Information from these job postings are extracted to create a database which contains the following types of information: 1) the job title, 2) the company, 3) the recruiter, 4) the location, 5) the salary, 6) the languages and platforms used, and 7) required or desired years of experience and degrees (Califf & Mooney, 1999).

The U.S. Geography domain has a corpus of 880 sentences, in which 250 were collected from undergraduate students in our department and the rest from real users of our Web interface GEOQUERY at `www.cs.utexas.edu/users/ml/geo.html` over a period of approximately one to two years. The job database information system has a corpus of 640 sentences; 400 of which are artificially made from a simple grammar that generates certain obvious types of questions people will ask and the other 240 are questions obtained from real users of our interface JOBFINDER. Both corpora are available at the ftp site: `ftp.cs.utexas.edu/pub/mooney/nl-ilp-data/`.

4.3.2 Experimental Design

The experiments were conducted using 10-fold cross validation. In each test, the recall (a.k.a. accuracy) and the precision of the parser are reported. Recall and precision are defined as

$$Recall = \frac{\text{\# of correct queries produced}}{\text{\# of sentences}} \tag{4.7}$$

$$Precision = \frac{\text{\# of correct queries produced}}{\text{\# of successful parses}}. \tag{4.8}$$

The recall is the number of correct queries produced divided by the total number of sentences in the test set. The precision is the number of correct queries produced divided by the number of sentences in the test set from which the parser produced a query (i.e. a successful parse). Please note that a query is considered correct if it produces the same answer set as that of the correct logical query.

In information extraction, recall is usually defined as $|I|/|C|$ where I is the intersection of the set of retrieved documents R (from executing the user query) and the set of correct documents C. Precision is usually defined as $|I|/|R|$. Here, in semantic parsing, recall is defined as $|I|/|C|$ where I is the intersection of the set S of logical queries produced by the parser in parsing the set of test sentences and C which is the set of logical queries, one for each sentence, in the given set of test sentences. Precision is defined as $|I|/|S|$. In other words, recall is simply the fraction of test sentences that were correctly parsed by the learned parser (i.e. by producing a correct logical query for the sentence) and precision is the probability that the logical query produced by the learned parser from parsing a test sentence is correct.

Parser \ Corpora	Geo880				Jobs640			
	R	P	S	T	R	P	S	T
COCKTAIL	79.40	89.92	64.79	62.88	79.84	93.25	105.77	68.10
f_{mFOIL} only	75.10	88.98	127.61	76.67	63.75	82.26	427.02	66.64
$f_{CHILLIN}$ only	70.80	91.38	150.69	41.24	72.50	86.24	177.99	43.81
CHILLIN	71.00	90.79	142.41	38.24	74.22	87.48	175.94	45.31
mFOIL	67.50	87.10	204.62	65.17	58.91	82.68	561.34	69.08

Table 4.1: Results on all the experiments performed. Geo880 consists of 880 sentences from the U.S. Geography domain. Jobs640 consists of 640 sentences from the job postings domain. COCKTAIL is using both the f_{mFOIL} and the $f_{CHILLIN}$ clause constructors. f_{mFOIL} only and $f_{CHILLIN}$ only are COCKTAIL using just the single clause constructor only. R = recall, P = precision, S = average size of a hypothesis found for each induction problem when learning a parser using the entire corpus, and T = average training time in minutes.

4.3.3 Results and Discussion

COCKTAIL (f_{mFoil}+$f_{Chillin}$) is COCKTAIL using both the clause constructors from CHILLIN and mFOIL. COCKTAIL (f_{mFoil} only) is COCKTAIL using only the clause constructor from mFOIL. COCKTAIL ($f_{Chillin}$ only) is COCKTAIL using only the CHILLIN's clause constructor.

For all the experiments performed, we used a beam size of four for mFOIL (and therefore for f_{mFoil}), a significant threshold of 6.64 (i.e. 99% level of significance), and a parameter $m = 10$. We took the best four clauses (by coverage) found by CHILLIN. COCKTAIL using both mFOIL's and CHILLIN's clause constructors performed the best; it outperformed all other learners by at least 4% in recall in both domains. We performed the following one tail paired t-tests for the recall to see if the differences are significant: 1) In Geo880, COCKTAIL (f_{mFoil}+$f_{Chillin}$) vs. COCKTAIL ($f_{Chillin}$ only), 2) In Geo880, COCKTAIL (f_{mFoil}+$f_{Chillin}$) vs. COCKTAIL (f_{mFoil} only), 3) In Jobs640, COCKTAIL (f_{mFoil}+$f_{Chillin}$) vs. COCKTAIL

($f_{Chillin}$ only), and 4) In Jobs640, COCKTAIL (f_{mFoil}+$f_{Chillin}$) vs. COCKTAIL (f_{mFoil} only). The t-test results are as follows: 1) $t(10) = 8.714193, p = 5.55 \times 10^{-6}$, 2) $t(10) = 11.19033, p = 6.96 \times 10^{-7}$, 3) $t(10) = 5.560667, p = 0.000176$, and 4) $t(10) = 12.03153, p = 3.76 \times 10^{-7}$. In all these results, $p < 0.05$. We can say that they were significant results.

In addition, COCKTAIL using only f_{mFoil} performed better than using only $f_{Chillin}$ in the Geography domain while the latter performed better in the job postings domain. This indicates that there is a heavier bias on learning specific contextual information (like a specific word or phrase in the input buffer) (by f_{mFoil}) in the Geography domain while in the job postings domain there is a heavier bias on learning structural features (by $f_{Chillin}$) for disambiguation. Notice that CHILLIN alone performed slightly better than COCKTAIL using only $f_{Chillin}$. There must be other factors in the picture we were not aware of as using a hill-climbing search actually performed better in this case. One possibility might be that the current MDL metric has problems handling complicated terms with lots of constants which could result in choosing overly specific clauses (if they are in the beam) and therefore learning a larger number of clauses for the building theory. Perhaps somewhat surprising is the result obtained from using the original mFOIL algorithm; the poor results seem to suggest that choosing the most statistically significant clause (in the search beam) does not necessarily produce the most compressive hypothesis. Apparently, this is due to the fact that some compressive clauses were wrongly rejected by a statistical based measure, which is a problem of using a statistical based search heuristic (e.g. computing the statistical significance of a clause) versus a compression based heuristic (e.g. computing the complexity of a clause) as reported in (Muggleton et al., 1992).

COCKTAIL (f_{mFoil}+$f_{Chillin}$) also found the most compressive hypothesis on average in both domains; COCKTAIL (f_{mFoil}+$f_{Chillin}$)'s hypothesis was at most half of the size of that of all other hypotheses found by other learners in Geo880, and it was at most 60% of the size of hypotheses found by other learners in Jobs640. COCKTAIL (f_{mFoil}+$f_{Chillin}$)'s training time was more than that of CHILLIN in both domains by roughly 20 minutes of CPU time. CHILLIN was the fastest in training since it uses a hill-climbing search to find a good clause while all other learners use beam search.

mFOIL and CHILLIN learn very different features for classification; mFOIL is given background predicates which check the presence (or absence) of a particular element in a given parse state (e.g. a certain predicate or a certain word phrase) while CHILLIN is not given any such background predicates but it learns the structural features of a parse state through finding *LGGs* with good coverage (and inventing predicates if necessary). Each learner is effective in expressing each type of feature using its own language bias; if one were to learn structural features of a parse state using mFOIL's language bias (e.g. not allowing function terms), the hypothesis thus expressed would have a very high complexity and vice versa.

In a nutshell, the problem of semantic parser acquisition requires learning two types of parsing features for disambiguation: 1) the context of a parse state, and 2) its functional structure A top-down ILP approach is better for learning the former while a bottom-up approach is better for the latter. Combining both ILP approaches can outperform CHILLIN (the previous leading ILP system), which was shown in our experimental results.

Chapter 5

Relational Data Mining: Pattern Learning in Link Discovery

5.1 Introduction

The terrible events of September 11, 2001 have sparked increased development of information technology that can aid intelligence agencies in detecting and preventing terrorism. The Evidence Extraction and Link Discovery (EELD) program of the Defense Advanced Research Projects Agency (DARPA) is one attempt to develop new computational methods for addressing this problem. More precisely, *Link Discovery* (LD) is the task of identifying known, complex, multi-relational patterns that indicate potentially threatening activities in large amounts of relational data. Some of the input data for LD comes from *Evidence Extraction* (EE), which is the task of obtaining structured evidence data from unstructured, natural-language documents (e.g. news reports), other input data comes from existing relational databases (e.g. financial and other transaction data). Finally, *Pattern Learning* (PL) concerns the automated discovery of new relational patterns for detecting potentially threatening activities in large amounts of multi-relational data.

Domain	*# Bg. preds.*	*Avg. Arity*	*# Bg. facts*
Link Discovery	52	2	≈ 568k
Bioinformatics	36	4.9	≈ 24k

Table 5.1: Link Discovery versus Bioinformatics (e.g. carcinogenesis). *# Bg. preds.* is the number of different predicate names in the background knowledge, *Avg. Arity* is the average arity of the background predicates, and *# Bg facts* is the total number of ground background facts.

Scaling to large datasets in data mining typically refers to increasing the *number* of training examples that can be processed. Another measure of complexity that is particularly relevant in multi-relational data mining is the *size* of examples, by which we mean the number of ground facts used to describe the examples. To our knowledge, the challenge problems developed for the EELD program are the largest ILP problems attempted to date in terms of the number of ground facts in the background knowledge. Relational data mining in bioinformatics (Zelezny, Srinivasan, & Page, 2002), e.g. carcinogenesis, was probably the previously largest ILP problem in this sense. Table 5.1 shows a comparison between link discovery and, to our knowledge, the largest problem in bioinformatics.

Scaling up ILP to efficiently process large examples like those encountered in EELD is a significant problem. Section 5.2 discusses the problems existing ILP algorithms have scaling to large examples and presents our general approach to controlling the search for multi-relational patterns by integrating top-down and bottom-up search. Section 5.3 presents the details of our new algorithm, BETH. Section 5.4 presents some theoretical results on our approach. Experimental results are presented and discussed in Section 5.5.

5.2 Combining Top-down and Bottom-up Approaches in BETH

One problem with an approach like PROGOL is that given a positive example and background knowledge, the bottom clause can be infinite, and practically one has to bound it. In PROGOL, it is bounded by five parameters: i, r, $\mathcal{M}$, j^-, and j^+ (please refer to (Muggleton, 1995) or Chapter 2 for more details). Unfortunately, the complexity of PROGOL's bottom clause is exponential w.r.t. the variable depth i, which results in a hypothesis space that is doubly exponential! (The size of the subsumption lattice is two to the power of the size of the bottom clause.)

In problems with large examples like EELD, the background knowledge contains many facts using numerous predicates that describe each complex object or event. Typically, many of these facts are irrelevant to the task. However, PROGOL's bottom clause includes every piece of background knowledge (within the recall bound r) in its body. This leads to intractably large bottom-clauses which generates an exponentially larger hypothesis space when learning a clause. This leads one to wonder if it is possible to bound the bottom-clause differently so that it contains only a relevant subset of background facts.

A strength of the top-down approach is that the generation of literals is inherently directed by the heuristic search process itself: only the set of literals that make refinements to clauses in the search beam are generated. Clauses with insufficient heuristic value are discarded, saving the need to generate literals for them. So, there is a tangible link between the entire set of literals that could be included in a bottom-clause and the heuristic search for a good clause. Therefore, perhaps it is possible to employ the heuristic search as a guide to selecting a relevant subset of background facts for inclusion in an alternative bottom-clause.

A major weakness of the top-down approach (as far as literal generation is concerned) is that the enumeration of all possible combination of variables generates many more literals than necessary; some literals generated by the algorithm are not even guaranteed to cover one positive example. The complexity of enumerating all such combinations in FOIL (Quinlan, 1990) (and mFOIL (Lavrac & Dzeroski, 1994a)) is exponential w.r.t. the arity of the predicates (Pazzani & Kibler, 1992). [1] A corresponding strength of the bottom-up approach is that a literal is created using a ground atom describing a known positive example. The advantages are: 1) specializing using this literal results in a clause that is guaranteed to cover at least the seed example, and 2) the set of literals generated are constrained to those that satisfy 1).

Given the strengths and weaknesses of typical top-down and bottom-up approaches, it seems that one can take advantage of the strength of each approach by combining them into one coherent approach. More precisely, we no longer build the bottom clause using a random seed example before we start searching for a good clause. Instead, after a seed example is chosen, one generates literals in a top-down

[1] Enforcing argument type restrictions can help lower the complexity but cannot completely solve the problem.

fashion (i.e. guided by heuristic search) except that the literals generated are constrained to those that cover the seed example. Based on this idea, we have developed a new system called **B**ottom-clause **E**xploration **T**hrough **H**euristic-search (BETH) in which the bottom clause is not constructed in advance but "discovered" during the search for a good clause.[2]

5.3 The Algorithm

BETH's bottom clause is virtual in the sense that the algorithm does not have to construct it to work, unlike PROGOL/ALEPH; it is, nonetheless, constructed to facilitate collection of statistics. However, the virtual bottom clause is a real bound on the subsumption lattice (see Section 5.4).

5.3.1 Constructing a Clause

The outermost loop of BETH is a simple set covering algorithm like that of any typical ILP algorithm: 1) find a good clause which covers a non-empty subset of positive examples, 2) remove the positive examples covered by the clause from the entire set of positive examples, 3) add the clause found to the set of clauses being built (a.k.a. theory) which was initially empty, 4) repeat step 1) to step 3) until the entire set of positive examples are covered by the theory, 5) return the entire set of clauses found.

The way a clause is constructed in BETH is very similar to a traditional top down ILP algorithm like FOIL; the search for a good clause goes from general to specific. It starts with the most general clause $\square$ which is specialized by adding a

[2]PROGOL and ALEPH are really, more precisely, "Subsumption lattice exploration through heuristic-search". Here, we explore the bottom clause and the subsumption lattice simultaneously.

literal to its body. The most general clause $\Box = T \leftarrow true$ where T is a literal such that $predname(T) = predname(e)$ and $arity(T) = arity(e)$, where e is a randomly chosen seed example from the set of positive examples. We used beam search to find a sufficiently good clause.

In addition, we also compute the bottom clause which bounds the search space. The initial bottom clause is set to $e \leftarrow true$ where e, the seed example, is randomly chosen from the set of positive examples. The bound is expanded incrementally during the search for a good clause. The bound is fixed when a *sufficiently good* clause is found, at which point both the clause and the bound are returned as solutions to the search. The algorithm which constructs a clause is outlined in Figure 5.1.

---

1. Given a set of predicate specifications $\mathcal{P}$ of the background predicates, beam width b, clause length bound n, variable depth bound i, recall bound r, a non-empty set of positive examples Pxs, and a set (possibly empty) of negative examples Nxs.
2. Randomly choose a seed example $e \in Pxs$.
3. $\perp_0 := e \leftarrow true$.
4. $Q_0 := \{\Box\}$.
5. $Q := Q_0$.
6. $\perp := \perp_0$.
7. REPEAT
   $generate\_refinements(Q, \mathcal{P}, b, n, i, r, Pxs, Nxs, Q', \perp, \perp')$,
   $Q := Q'$,
   $\perp := \perp'$
   UNTIL there is a clause $C \in Q$ which is *sufficiently* accurate.
   (Q' and $\perp'$ are output variables and the rest in *generate_refinements* are input variables.)
8. Return C and $\perp$.

Figure 5.1: The construction of a clause in BETH

5.3.2 Generating Refinements for a Clause

To find all the refinements of a given clause C_i, first find a substitution θ that satisfies the body of the clause (a.k.a. "a successful proof" of the clause); then construct a literal (with dummy variables) R_j for a predicate specification in $\mathcal{P}$, and find a substitution β that makes $R_j\beta$ a ground atom such that $C_i\theta$ and $R_j\beta$ satisfy the following conditions we call *refinement constraints*: 1) the link constraint: one of the arguments of $R_j\beta$ has to appear in $C_i\theta$ (this is to make sure that the resulting clause is still a linked clause), 2) the unique-literal constraint: $R_j\beta \notin C_i\theta$ (to avoid making two identical literals). We try to find pairs of θ and β satisfying the refinement constraints, but at most r distinct ground atoms $R_j\beta$ will be used.

To avoid repeatedly finding a successful proof of a given clause by theorem proving, we make a set of "cached proofs" for each clause in the beam (similar to the way variable bindings are stored in extensional FOIL) by starting with the initial proof $e \leftarrow true$, where e is a randomly chosen seed example, and we incrementally update the cache of proofs of each clause by adding to the end of each proof a ground atom satisfying all the refinement constraints. A bound is also given to the cache size. When finding a satisfying substitution θ for a clause C_i in the beam, we will simply unify C_i with a proof in its cache. If there is no $R_j\beta$ satisfying the refinement constraints, which can happen if the first chosen example e was a "bad" one, a new example $e' \neq e$ will be randomly chosen from the remaining set of positive examples to be covered. The clause C_i will be replaced (in the beam) by the most general clause such that its cache of proofs will contain only $e' \leftarrow true$. The idea is that if a clause cannot be refined, then we will just restart with a different seed example.

One can take advantage of type declarations (if available or otherwise can be omitted) to further restrict the number of predicate specifications needed to be considered for a given clause (which is similar to Aleph's mode declaration).

One can also make use of input-output mode declarations (if available) by substituting arguments with "input" mode for constants which appear in the clause provided that the argument type and the constant type are compatible (similar to the algorithm that builds the bottom clause for PROGOL). In this case, one needs to find satisfying substitutions for R_j for each unique way of substituting arguments with input mode for constants in the clause. The algorithm of generating refinements to a clause is outlined in Figure 5.2.

---

1. Given a set of predicate specifications $\mathcal{P}$ of the background predicates, a beam width b, a bound on the clause length n, variable depth bound i, recall bound r, a non-empty set of positive examples Pxs and a set (possibly empty) of negative examples Nxs, the current bottom clause $\perp$ (i.e. the current bound on the search space).
2. For each clause $C_i \in Q$ and for each $P_j \in \mathcal{P}$, make a literal R_j with dummy variables such that $predname(R_j) = predname(P_j)$ and $arity(R_j) = arity(P_j)$.
3. Find substitutions θ, β such that 1) θ satisfies C_i, 2) β satisfies R_j, and 3) $C_i\theta$ and $R_j\beta$ satisfy all the *refinement constraints*.
4. Collect at most r such ground atoms $R_j\beta$ for different θ and β.
5. For each pair of $C_i\theta$ and $R_j\beta$ satisfying all the refinement constraints, $make\_literals(C_i\theta, R_j\beta, Lits)$ and add $R_j\beta$ to the body of $\perp$.
6. For each $L \in Lits$, add L to the body of C_i to make C_i' and let the set of all C_i''s be Q_i.
7. Evaluate each clause in $\bigcup_{C_i \in Q} C_i$ by a heuristic (e.g. m-estimate) given Pxs and Nxs.
8. Put only the best b clauses into Q'.
9. Let $\perp'$ be the resulting bottom clause after adding all the ground atoms $R_j\beta$'s to the body of $\perp$ for each C_i and R_j (such that there exists θ and β satisfying all the refinement constraints).
10. Return Q' and $\perp'$.

Figure 5.2: Generate Refinements

5.3.3 Making Literals

The idea behind making literals given a clause C, a satisfying substitution θ of the clause, and a particular ground atom $R_j\beta$ is that we want to replace arguments in the ground atom by variables in the clause which are instantiated (in θ) to these arguments in the ground atom, *provided that* the ground atom is not already part of $C\theta$ and the resulting literal observes the restriction on the variable depth bound. The algorithm for making literals is outlined in Figure 5.3.

---

1. Given a clause $C\theta$ of the form $e \leftarrow a_1, \ldots, a_n$ (where C is the current clause being refined, i.e. specialized, and θ is a substitution that satisfies C and $e \in Pxs$ and background knowledge $BK \models a_i$ for each ground atom a_i in the body of $C\theta$) and a ground atom a_{n+1} such that $BK \models a_{n+1}$.
2. Make a set of literals *Lits* such that each literal $L \in Lits$ satisfies: 1) $predname(L) = predname(a_{n+1})$, 2) $arity(L) = arity(a_{n+1})$, 3) suppose the constant c_i is the ith argument of a_{n+1} and the variable V_i is the ith argument of L. If c_i appears in $C\theta$, then $c_i/V_i \in \theta^{-1}$; otherwise, V_i is a new variable not appearing in C, 4) there is no variable V in L such that $d(V) > i$ where i is the variable depth bound.
3. Return *Lits*.

Figure 5.3: Make Literals

5.3.4 A Concrete Example

We can see how the algorithm works through a simple example from the family-relation domain. Suppose we want to learn the concept $uncle(X, Y)$, which is true iff X is an uncle of Y (blood uncle).

Suppose we have the following set of background facts (Figure 5.4):

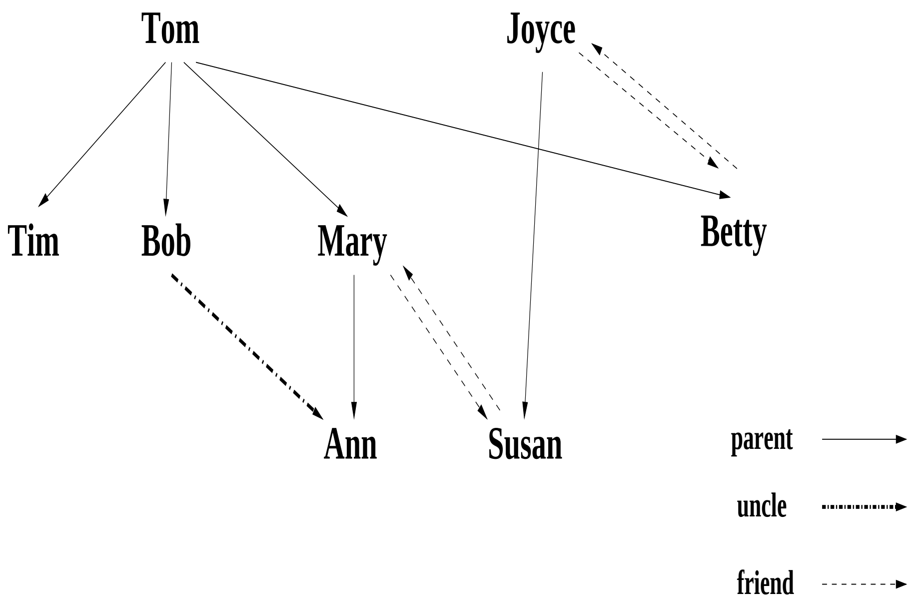


Figure 5.4: A simple family relation domain

1. $male(Bob)$, $male(Tom)$, $male(Tim)$
2. $female(Ann)$, $female(Mary)$, $female(Susan)$, $female(Betty)$, $female(Joyce)$
3. $parent(Tom, Mary)$, $parent(Tom, Betty)$, $parent(Tom, Bob)$, $parent(Mary, Ann)$,$parent(Joyce, Susan)$,$parent(Tom, Tim)$
4. $friend(Mary, Susan)$, $friend(Susan, Mary)$, $friend(Joyce, Betty)$, $friend(Betty, Joyce)$

and $\mathcal{P} = \{male/1, female/1, friend/2, parent/2\}$ (exactly in this order from left to right) is our set of predicate specifications. We will use $'+'$ to denote the output mode and $'-'$ the input mode here. The following is the set of mode specifications for each predicate specification:

$$male(-), female(-), parent(+,-), parent(-,+), friend(+,-), friend(-,+)$$

and the following is the set of type specifications for each predicate specification:

1. $male(person)$
2. $female(person)$
3. $parent(person, person)$
4. $friend(person, person)$

Suppose we have this set of training examples:

1. Positive: $uncle(Bob, Ann)$
2. Negative:
   $uncle(Bob, Susan)$, $uncle(Betty, Ann)$, $uncle(Tim, Susan)$, $uncle(Tom, Betty)$
   $uncle(Susan, Betty)$, $uncle(Joyce, Ann)$, $uncle(Tim, Joyce)$, $uncle(Tom, Mary)$

We present a trace of how our algorithm discovers a good clause, given a beam size and a recall bound of one, and a clause length of four. It starts by choosing a random seed example from the set of positive examples. This has to be $uncle(Bob, Ann)$ since there is only one positive example. When generating refinements to a clause, it considers each predicate specification in $\mathcal{P}$ (from left to right). We will show the specialized clause before its set of cached proofs. The literal added to the clause currently being built is generated from the *new* ground atom added to the body of the cached proof of the current clause.

The algorithm starts with:

1. The most general clause which covers every pair of people: `uncle(X,Y) :- true`

2. The set of cached proofs for this clause: {`uncle(bob,ann) :- true`}

3. The empty bottom clause: `uncle(bob,ann) :- true`

It considers $male/1$ and generates the following:

1. The specialized clause: `uncle(X,Y) :- male(X)`
   `(m-est = 0.153)`

2. The set of cached proofs for this clause: {`uncle(bob,ann) :- male(bob)`}

3. The updated bottom clause: `uncle(bob,ann) :- male(bob)`

Next, the algorithm considers $female/1$, and the literal `female(Y)` is generated (in the same way as $male/1$), the new ground atom `female(ann)` is added to the current bottom clause. The specialized clause `uncle(X,Y) :- female(Y)` has an m-estimate of 0.111. Next, it considers $parent/2$ (using $parent(+,-)$) and generates the following:

1. The specialized clause: `uncle(X,Y) :- parent(Z,X)`
   `(m-est = 0.136)`

2. The set of cached proof of this clause: {`uncle(bob,ann) :- parent(tom,bob)`}

3. The updated bottom clause:
   `uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob)`

Similarly $parent/2$ (using $parent(+,-)$) is used to generate another specialized clause `uncle(X,Y) :- parent(W,Y)` (m-estimate $= 0.122$) using the ground atom `parent(mary,ann)`.

The predicate specification $friend/2$ was considered but no ground atom was found to satisfy all the refinement constraints; the link constraint could not be satisfied, because neither Bob nor Ann has a friend. There are totally four different refinements to the most general clause. The clause with the best m-estimate is:

$$uncle(X, Y) \leftarrow male(X)$$

Since the beam size is just one, only this clause is retained in the beam. This clause is still covering negative examples: $uncle(Bob, Susan)$, $uncle(Tom, Betty)$, $uncle(Tim, Susan)$, $uncle(Tim, Joyce)$, and $uncle(Tom, Mary)$. So, it still needs to be refined. Next, $male/1$ is considered but no ground atom is found to satisfy all the refinement constraints; the unique-literal constraint could not be satisfied ($male(Bob)$ is already in the cached proof of the clause). The current bottom clause is `uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob)`.

Next, it considers $female/1$ and generates the following:

1. The specialized clause: `uncle(X,Y) :- male(X),female(Y)` `(m-est = 0.153)`
2. The set of cached proof of this clause: {`uncle(bob,ann) :- male(bob),female(ann)`}
3. The updated bottom clause: `uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob), parent(mary,ann)`

Next, it considers $parent/2$ (using $parent(+,-)$) and generates the following:

1. The specialized clause: `uncle(X,Y) :- male(X),parent(Z,X)` `(m-est = 0.204)`
2. The set of cached proof of this clause: {`uncle(bob,ann) :- male(bob),parent(tom,bob)`}
3. The updated bottom clause: `uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob),`

```
parent(mary,ann)
```

$parent/2$ (using $parent(+,-)$) can be used to generate another specialized clause `uncle(X,Y) :- male(X),parent(W,Y)` (m-estimate $= 0.175$) using the ground atom `parent(mary,ann)`.

The predicate specification $friend/2$ was considered but no ground atom was found to satisfy all the refinement constraints (the link constraint cannot be satisfied). There are totally three different refinements to $uncle(X,Y) \leftarrow male(X)$. The clause with the best m-estimate is:

$$uncle(X,Y) \leftarrow male(X), parent(Z,X)$$

This clause still covers a non-empty set of negative examples:

$$uncle(Bob, Susan), uncle(Tim, Susan), uncle(Tim, Joyce).$$

The algorithm continues in exactly the same manner for the last two steps. The clause $uncle(X,Y) \leftarrow male(X), parent(Z,X)$ has four different refinements. The clause with the best m-estimate is:

$$uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y)$$

which is still covering negative examples $uncle(Bob, Susan)$ and $uncle(Tim, Susan)$.

There are totally eight different refinements to

$$uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y).$$

The clause with the best m-estimate is:

$$uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y), parent(Z,W)$$

which covers all the positive examples and no negative examples. At this point, the algorithm has found the target concept. Both the bottom clause discovered and the consistent clause found are returned. The bottom clause found by BETH is:

```
uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob),male(tom),
parent(mary,ann),female(mary),parent(tom,mary),friend(mary,susan),
friend(susan,mary)
```

whereas, PROGOL's bottom clause is:

```
uncle(bob,ann) :- male(bob),female(ann),parent(tom,bob),male(tom),
parent(mary,ann),female(mary),parent(tom,mary),friend(mary,susan),
friend(susan,mary),female(susan)
```

which is bigger than BETH's bottom clause by just one ground literal in this simple domain. Note that the ground literal `female(susan)` is part of PROGOL's bottom clause because the constant `susan` is in the set $InTerms$ (see Section 2.5.4).

5.4 Analysis of Algorithm

We will present result on the complexity of the bottom clause constructed by BETH which is only linear compared to that of PROGOL which is exponential. Since the size of the hypothesis space is two to the power of the size of the bottom clause, the hypothesis space is doubly exponential in complexity if one searches the subsumption lattice (i.e. the hypothesis space) bounded by PROGOL's bottom clause.

Let $\perp(b, n, \mathcal{P}, r, i)$ be the bottom clause constructed by the algorithm outlined in Section 5.3 given the parameters b, n, $\mathcal{P}$, r, and i which are the beam width, the maximum clause length, the set of predicate specifications, the recall bound, and the variable depth bound respectively.

Lemma 3. $i \leq n$ where i is the maximum variable depth of any variable in a linked clause C and n is the length of C.

Proof. We can prove by induction on n. Base Case: $n = 0$. In this case, the body of the clause is empty. $d(V) = 0$ for any variable V which appears in the head of the clause. So, we have $i \leq n$. Inductive Case: assume that for a given linked clause of length k, $i \leq k$. Now, suppose $C : H \leftarrow L_1, \ldots, L_{k+1}$ is a linked clause of length $k+1$ and i is the maximum variable depth of C. Therefore, the clause $C' : H \leftarrow L_1, \ldots, L_k$ is a linked clause of length k and so we have $i' \leq k$ where i' is the maximum variable depth of any variable in C'. Case 1: There is no variable V in the literal L_{k+1} s.t. V does not appear in C'. So, $i = i'$ and we have $i \leq k+1$. Case 2: There is a variable V in L_{k+1} s.t. V does not appear in C'. From the definition of variable depth (Lavrac & Dzeroski, 1994a), we have $d(V) \leq i' + 1$. So, we have $i \leq k + 1$. By M.I., we have proved the lemma. □

Theorem 4. Suppose B is a beam of clauses produced by BETH, for any clause $C \in B$, $C \preceq \perp(b, n, \mathcal{P}, r, i)$.

Proof. Suppose C_j is a clause in B such that $C_j = H \leftarrow L_1, \ldots, L_m$ where $m \leq n$. Each L_k is produced from some ground atom a_k and H is produced from a particular seed example e. Obviously, $e \in \perp(b, n, \mathcal{P}, r, i)$. For any $k : 1 \leq k \leq m$, $a_k \in \perp(b, n, \mathcal{P}, r, i)$, since each ground atom which satisfies all the refinement constraints is added to the current bottom clause and only ground atoms satisfying all the refinement constraints are used to make literals for any given clause. Therefore, there exists a substitution θ which satisfies C_j such that $C_j\theta = e \leftarrow a_1, \ldots, a_m$. Therefore, $C_j\theta \subseteq \perp(b, n, \mathcal{P}, r, i)$. In other words, we have $C_j \preceq \perp(b, n, \mathcal{P}, r, i)$. Hence we have $C \preceq \perp(b, n, \mathcal{P}, r, i)$ for any clause $C \in B$. □

Theorem 5. The worst case length of $\perp(b, n, \mathcal{P}, r, i)$ is $\mathcal{O}(bn|\mathcal{P}|r)$.

Proof. The maximum number of ground atoms that 1) satisfy the refinement constraints and 2) make literals observing the variable depth bound i for any clause in the search beam at the point the clause is being refined are $|\mathcal{P}|r$. Therefore, the maximum number of ground atoms satisfying the refinement constraints after adding n literals to the body of the most general clause are $n|\mathcal{P}|r$. Since there are at most b clauses in the search beam at any time, the maximum number of ground atoms satisfying the refinement constraints are $bn|\mathcal{P}|r$. Thus, the worst case complexity of the bottom clause $\perp(b, n, \mathcal{P}, r, i)$ is $\mathcal{O}(bn|\mathcal{P}|r)$. □

The length of PROGOL's bottom clause is $\mathcal{O}((r|\mathcal{M}|j^+j^-)^{ij^+})$ (Muggleton, 1995) (which makes a hypothesis space doubly exponential w.r.t. i) while the length of BETH's bottom clause is only linear w.r.t. $n \geq i$ (which gives rise to a much smaller hypothesis space).

5.5 Experimental Evaluation

5.5.1 Experimental Setup

ALEPH represents the state-of-the-art ILP system based on the concept of inverse entailment which, to a certain extent, has its origin in bottom-up ILP approaches. ALEPH is also a hybrid ILP approach in the sense that it employs a FOIL-like refinement operator (i.e. it searches for a good clause in a general to specific manner). mFOIL (or FOIL for that matter) is the best (purely) top-down ILP approach. It is included in our experiments as a "control" factor to show the real "margin" of advantage gained by a hybrid ILP system. BETH combines the best of PROGOL and the best of mFoil like we mentioned in Section 5.2. We are going to have a "head-to-head" comparison among the three ILP systems.

5.5.2 Domain

After the events of 9/11, the EELD project has been working on several Challenge Problems that are related to counter-terrorism. The problem that we choose to tackle is the detection of Murder-For-Hires (contract killings) in the domain of Russian Organized Crime. The data used in all EELD Challenge Problems include representations of people, organizations, objects, and actions and many types of relations between them. One can picture this data as a large graph of entities connected by a variety of relations. For our purposes, we represent these relational databases as facts in Prolog.

For the ease of generating large quantities of data, and to avoid violating privacy, the program currently only uses synthetic data generated by a simulator. The data for the Murder-For-Hire problem was generated using a Task-Based (TB) simulator developed by Information Extraction and Transport Incorporated (IET). The TB simulator outputs case files, which contain complete and unadulterated descriptions of murder cases. These case files are then filtered for observability, so that facts that would not be accessible to an investigator are eliminated. To make the task more realistic, this data is also corrupted, e.g., by misidentifying role players or incorrectly reporting group memberships. This filtered and corrupted data form the evidence files. In the evidence files, facts about each event are represented as ground facts, such as:

```
murder(Murder714)
```

```
perpetrator(Murder714, Killer186)
crimeVictim(Murder714, MurderVictim996)
deviceTypeUsed(Murder714, PistolCzech)
```

The synthetic dataset that we used consists of 632 murder events. Each murder event has been labeled as either a positive or negative example of a murder-for-hire. There are 133 positive and 499 negative examples in the dataset. Our task was to learn a theory to correctly classify an unlabeled event as either a positive or negative instance of murder-for-hire. The amount of background knowledge for this dataset is extremely large; consisting of 52 distinct predicate names, and 681,039 background facts in all.

5.5.3 Results

The performance of each of the ILP systems was evaluated using 6-fold cross-validation. The total number of Prolog atoms in the data is so large that running more than six folds is not feasible.[3] The data for each fold was generated by separate runs of the TB simulator. The facts produced by one run of the simulator, only pertain to the entities and relations generated in that run; hence the facts of each fold are unrelated to the others. For each trial, one fold is set aside for testing, while the remaining data is *combined* for training. To test performance on varying amounts of training data, learning curves were generated by testing the system after training on increasing subsets of the overall training data. Note that, for different points on the learning curve, the background knowledge remains the same; only the number of positive and negative training examples given to the system varies.

[3]The maximum number of atoms that the Sicstus Prolog compiler can handle is approximately a quarter million.

We compared the three systems with respect to accuracy and training time. Accuracy is defined as the number of correctly classified test cases divided by the total number of test cases. The training time is measured as the CPU time consumed during the training phase. All the experiments were performed on a 1.1 GHz Pentinum with dual processors and 2 GB of RAM. BETH and mFOIL were implemented in Sicstus Prolog version 3.8.5 and ALEPH was implemented in Yap version 4.3.22. Although different Prolog compilers were used, the Yap Prolog compiler has been demonstrated to outperform the Sicstus Prolog compiler, particularly in ILP applications (Santos Costa, 1999).

In our experiments, we used a beam width of 4 for BETH and mFOIL; and limited the number of search nodes in ALEPH to 5000. We used m-estimate ($m = 2$) as a search heuristic for all ILP algorithms. The clause length was limited to 10 and the variable depth bound to 5 for all systems. The recall bound was limited to 1 for BETH and ALEPH (except for some mode declarations it was set to '*'). We modified mFOIL to be constrained by the maximum clause length and the variable depth bound, to ensure that it terminates. We refer to this version of mFOIL as *Bounded* mFOIL. All the systems were given 1 second of CPU time to compute the set of examples covered by a clause. If a specialized clause took more time than allotted, the clause was ignored; although the time it took to create the clause is still recorded.

The results of our experiments are summarized in Appendix F for test accuracy and training time. A snapshot of the performance of the three ILP systems given 100% of the training examples is shown in Table 5.2. The following is a sample rule learned by BETH:

System	*Accuracy*	*CPU Time*	*# of Clauses*	$\perp$ *Size*	*Avg Theory Size*
BETH	94.80%	23.39	4483	34	125
ALEPH	96.91%	598.92	63334	4061	41
mFOIL	91.23%	45.28	112904	n/a	214

Table 5.2: Results on classifying *murder-for-hire* events given all the training data. CPU time is in minutes; and *# of Clauses* is the total number of clauses tested; and $\perp$ *Size* is the average number of literals in the bottom clause constructed for each clause in the learned theory; and *Avg Theory Size* is the average # of literals in the learned theory.

```
murder_for_hire(A):- murder(A), eventOccursAt(A,H),
  geographicalSubRegions(I,H), perpetrator(A,B),
  recipientOfinfo(C,B), senderOfinfo(C,D), socialParticipants(F,D),
  socialParticipants(F,G), payer(E,G), toPossessor(E,D).
```

This rule covered 9 positive examples and 3 negative examples. The rule can be interpreted as: A is a murder-for-hire, if A is a murder event, which occurs in a city in a subregion of Russia, and in which B is the perpetrator, who received information from D, who had a meeting with and received some money from G. Some sample theories learned by the three ILP systems can be found in appendix D.

5.5.4 Discussion of Results

On the full training set, BETH trains 25 times faster than ALEPH while losing only 2 percentage points in accuracy and it trains twice as fast as mFOIL while gaining 3 percentage points in accuracy. Therefore, we believe that its integration of top-down and bottom-up search is an effective approach to dealing with the problem of scaling ILP to large examples. The learning curves further illustrate that the training time of BETH grows slightly slower than that of mFOIL, and considerably slower than that of ALEPH.

The large speedup over ALEPH is explained by the theoretical analysis on the complexity of the bounds on the search space, i.e. the different sizes of the bottom clauses they construct. The size of the bottom clause for BETH is only linear w.r.t. n compared to that of ALEPH which is exponential w.r.t. to i ($i \leq n$) even for small i. As a result, ALEPH's search space is much larger than BETH's. ALEPH's bottom clause was on average 119x larger than BETH's and the total number of clauses it constructed was 14x larger, although a theory of similar accuracy was learned.

Systems like BETH and ALEPH construct literals based on actual ground atoms in the background knowledge, guaranteeing that the specialized clause covers at least the seed example. On the other hand, mFOIL generates more literals than necessary by enumerating all possible combination of variables. Some such combinations make useless literals; adding any of them to the body of the current clause makes specialized clauses that do not cover any positive examples, which happens even when the combination of variables is consistent with the type of each argument in the literal. Thus, mFOIL wastes CPU time constructing and testing these literals. Since the average predicate arity in the EELD data was small (2), the speedup over mFOIL was not as great, although much larger gains would be expected for data that contains predicates with higher arity.

The average size of a theory learned by ALEPH is the smallest; only one third of that of BETH and one fifth of that of mFOIL. Searching a much larger hypothesis space allows one to find a more compressive hypothesis, which explains why ALEPH performed slightly better than BETH and mFOIL, at the expense of spending significantly more CPU time in learning. From the experimental results, we can conclude that 1) an approach like Beth, which intelligently searches a hypothesis space, can outperform (in efficiency) an approach like Progol/Aleph, which searches a much larger hypothesis space only to learn a theory of similar performance, and 2) constructing clauses based on a seed example avoids testing useless literals, which improves training time significantly over a purely top-down approach like mFoil.

Chapter 6

Query Transformations

6.1 Transformation Algorithms

Like many other algorithms in the field of machine learning, ILP algorithms construct "hypotheses" for data by performing a search through a large space. Such a search typically involves generating and then testing the quality of candidates. To test the quality of a candidate hypothesis, one needs to first compute the set of positive and negative examples covered by it and then compute the heuristic value for it given the set of examples covered and a particular search heuristic (e.g. m-estimate).

In order to compute the set of positive (or negative) examples covered by a candidate hypothesis, one needs to be able to tell if a particular (training) example is covered. However, to find out if a training example is covered by the candidate hypothesis, one needs to first bind the head of the candidate clause [1] with the example, which produces a ground (or non-ground) literal called a *query* in logic

[1]The hypothesis space of ILP is a set of Horn clauses.

programming terminology. After that, one needs to execute (or prove) the query in PROLOG (i.e. first-order Horn clause logic) using the background knowledge, which leads to the issue of improving the efficiency of hypothesis evaluation in an ILP system since theorem proving is an NP hard problem in general (Cook, 1971). [2] However, there are circumstances under which the efficiency of theorem proving can be dramatically improved if the structure (e.g. the ordering of literals in the body of the clause since theorem proving in PROLOG is "from left to right") of the clause is appropriately transformed. Since one concerns developing transformation algorithms that can transform an original clause into a functionally equivalent clause that "faciliates" theorem proving so as to significantly boost the efficiency of query execution (in PROLOG), this area of research has, therefore, come to be called *query transformation.* [3]

We will first introduce two existing transformation algorithms: 1) the cut transformation and 2) the once transformation (Costa et al., 2002). The former was developed as a simple transformation while the latter was developed as a more full-fledged technique, which will be explained later in the chapter. Therefore, one usually employs the once transformation in an ILP system for better query execution efficiency. These two transformation algorithms have one thing in common: they take as input an entire clause and produce the transformed clause by processing every literal in the body of the clause. One shortcoming of this earlier approach is that every time a new candidate hyothesis is to be tested, the transformation algorithm has to start all over from the beginning of the clause only to repeat work

[2]It is, more precisely, semi-decidable but since we are only interested in tackling ILP problems that are decidable/solvable, the un-decidability issue is not of our concern here.

[3]Although it might be more appropriately called "clause transformation", we, nevertheless, decided to go with the established terminology.

that has been done before. For example, to evaluate the clause C, it has to be first transformed to, say, C'. After C' is evaluated, it will be discarded. Suppose the next thing the ILP system did was adding a literal L to the body of C to make $C1$. Similarly, $C1$ needs to be transformed first before its coverage on the set of training examples is computed. Now, to transform $C1$, which differs from C by one literal L, the transformation algorithm has to start from the very beginning of $C1$ only to repeat work that would have been done when transforming C.

This leads one to wonder if it might be possible to save some work done before by saving C' and re-using it in the process of transforming $C1$. This forms the basis of the idea behind my new approach I call *incremental* cut-and-once transformation, which takes as input a *transformed* clause C' (where C is the original clause), and a literal L to be added to the body of C, and produces a new transformed clause $C1'$ that is functionally equivalent to $C1$ (the result of adding L to the body of C). It is called *incremental* because the transformation starts with an already transformed clause (the current clause) and the result of transforming the specialized clause is, in a sense, like the already transformed clause "incremented" with the newly added literal. We will discuss the transformation algorithms in details here.

6.1.1 The Cut-transformation $t_!$

We observe that when executing a conjunction of goals $G_1, ..., G_n$, failure of a goal G_i will result in attempting to generate more solutions for goals earlier in the sequence. This effort is useless if these solutions do not alter the computation of G_i. The *cut-transformation*, $t_!$, exploits the notion of goal independence by partitioning the set of goals in a clause into classes such that goals in different classes are independent.

We will execute each class in sequence, and use the pruning operator, !, to avoid any backtracking between classes.

In pure logic programs, goals depend on each other because they share variables. Given a function $vars(T)$ that returns all variables in the term T, two goals G_i and G_j are said to *share*, that is the relation $Shares(G_i, G_j)$ holds, when:

$$i = j \ \vee \ vars(G_i) \cap vars(G_j) \neq \emptyset$$

The definition ensures that the relation $Shares$ is reflexive and symmetric. Its transitive closure $Linked$, defined as the smallest transitive relation that is a superset of $Shares$, is reflexive, symmetric and transitive, and therefore an equivalence relation.

Given a set of goals $\mathcal{G} = \{G_1, \ldots, G_n\}$, we shall name the equivalence classes established by $Linked$ as $I_1, \ldots, I_m$. If two goals are in the same equivalence class, we will say they are *dependent*, otherwise we will say they are *independent*. We would like to divide the original clause into several conjunctions of independent goals and execute them separately. To do so, we place the goals in each class I_i in a conjunction $\mathcal{G}_i$. Our notion of dependence is a safe approximation of the dependencies that can possibly occur at run-time, because as soon as two goals have a variable in common they belong to the same equivalence class. Moreover, the computation of the equivalence classes is efficient.

In this approximation, all goals that include a head variable or that share variables with one such goal will belong to the same equivalence class. Often, we know beforehand that head variables will be grounded before calling the body. Clearly, such variables cannot introduce sharing. To take advantage of this extra information, we classify variables as either $Grounded$ or $PossiblySharing$, and define $vars(T)$ to return only all $PossiblySharing$ variables in T.

To effect $t_!$ it is sufficient to implement the following procedure:

1. Given the original clause:

$$H\texttt{:-}\ G_1,\ \ \ldots,\ \ G_i,\ \ \ldots,\ \ G_n.$$

   Classify all variables in $H, G_1, \ldots, G_n$ as $PossiblySharing$ or $Grounded$. Compute the equivalence classes for the (approximated) sharing relation.

2. Number the goal literals according to the equivalence class they belong to:

$$H\texttt{:-}\ G_{1j},\ \ \ldots,\ \ G_{ik},\ \ \ldots,\ \ G_{nl}.$$

   where G_{ij} is the ith goal literal in equivalence class j.

3. Reorder the literals in the clause according to the class they belong to:

$$H\texttt{:-}\ G_{a1},\ \ \ldots,\ \ G_{b1},\ldots,\ \ G_{cm},\ \ \ldots,\ \ G_{dm}.$$

   where G_{ij} is the ith goal literal in equivalence class j.

4. We are interested in any solution, if one exists. Thus we need to compute every class once and if a class has no solution, the computation should wholly fail. The following program transformation puts a `cut` between each class to guarantee such a computation:

$$H\texttt{:-}\ G_{a1},\ \ \ldots,\ \ G_{b1},\ !\ ,\ldots,\ !\ ,\ \ G_{cm},\ \ \ldots,\ \ G_{dm}.$$

The transformation is correct in the sense that the examples derivable before and after the transformation are the same. Step (3) is correct because the switching lemma allows us to reorder goals. Step (4) is correct because whenever we introduce a new cut, all goals before the cut are independent of all goals after the cut. Backtracking to before the cut therefore could never result in new solutions for the goals after the cut.

Example 2 *Suppose we are given a clause* $C : f(A) \leftarrow g(A,X), h(X,Y), p(A,Z)$. *Since only the literals* $g(A,X)$ *and* $h(X,Y)$ *share a non-grounded (i.e. "possibly sharing") variable* X. *The literal* $p(A,Z)$ *shares no non-grounded variables with other literals in the body of* C. *Therefore, we only have two equivalence classes here: 1)* $g(A,X), h(X,Y)$ *and 2)* $p(A,Z)$. *The result of applying* $t_!$ *to* C *is the clause* $C' : f(A) \leftarrow g(A,X), h(X,Y), !, p(A,Z)$.

6.1.2 The Once-transformation t_o

The cut transformation just described ensures that the search for each independent set of subgoals always stops after finding the first solution. Notice that the transformed body goal $q_1, !, q_2, !, \ldots, q_n$ can be written as $once(q_1), once(q_2), \ldots, once(q_n)$ where the meta-predicate `once` is defined as

$$once(X) \texttt{:-}\ \ X,\ \ !.$$

which finds only one value of X that satisfies the goal. While at the clause level the `once` constructs have the same effect as the cuts, they have the advantage that they can flexibly "encapsulate" a particular group of literals within an equivalence class of literals, whereas cuts cannot. Example 3 below demonstrates the fundamental difference between the cut transformation and the once transformation.

The objective is to fine-tune $t_!$ by processing each set of independent goals in more detail. We can improve the partitioning process further by using extra data for each non-head variable. More precisely, we shall transform each independent set of goals by considering the variables grounded by the first literal just like the ground head variables in the cut transformation. We then apply the transformation recursively and refine each independent set of subgoals into subsubsets. We therefore 'look inside' each equivalence class returned by the $t_!$. First, we find a prefix of one or more literals such that each literal will ground some of its arguments, then apply the cut transformation on the rest of this subgoal, treating the grounded variables just like ground head variables.

This is the so called *once-transformation* or t_o. Blockeel et al. (Blockeel, Demoen, Janssens, Vandecasteele, & Laer, 2000) describe two different versions of t_o. The dynamic version transforms queries during their execution, which results in an overhead but which also makes it possible to check groundness and sharing during execution instead of having to pre-compute a safe approximation. However, we only present the static version, here.

For the static version, there is the open issue of how to estimate which literals ground or cause sharing between which arguments. It is reasonable to assume that such information is provided either by the user or through analysis. For instance, in many ILP data sets, most predicates are defined by ground facts or range-restricted clauses only; such predicates always ground all their arguments (and hence do not cause sharing).

A high level description of the once-transformation algorithm is shown in Figure 6.1. Essentially, the algorithm finds a prefix of a conjunction such that this prefix grounds enough variables for the rest of the conjunction to contain independent subgoals; then it is called recursively on these subgoals. Note that, similar to

once-transform(q):
  let $q = l_1, l_2, \ldots, l_n$
  find the smallest k s.t. there exists a partition $\mathcal{P}$ of $\{l_{k+1}, \ldots, l_n\}$ s. t.
  $\forall q_i, q_j \in \mathcal{P}$:
    $V(q_i) \cap V(q_j) \subseteq GroundedVars(\{l_1, \ldots, l_k\})$ and
    $\forall v \in V(q_i), w \in V(q_j) : \{v, w\} \notin PossiblySharing(\{l_1, \ldots, l_k\})$
  for all $q_i \in \mathcal{P}$: once-transform(q_i)

Figure 6.1: The once-transformation algorithm. $GroundedVars(G)$ contains all variables grounded by a call to G. $PossiblySharing(G)$ is the set of all pairs of variables that may share after a call to G.

the cut transformation, we assume that the order of independent groups of literals may be switched but the relative order of literals that are dependent of each other does not change.

Example 3 *Given* $C : t(A) \leftarrow s(A,X,Y), b(Y,Z), c(Z), d(X,U), e(U), p(A,W)$. $t(A) \leftarrow p(A,W), !, s(A,X,Y), once(b(Y,Z), c(Z)), once(d(X,U), e(U))$ *is the result of applying* t_o *to* C. *There are two independent subgoals in the body of the transformed clause:* $p(A,W)$, *and* $s(A,X,Y), once(b(Y,Z), c(Z)), once(d(X,U), e(U))$. *The clause* $t(X) \leftarrow p(A,W), !, s(A,X,Y), b(Y,Z), c(Z), d(X,U), e(U)$ *is the result of applying the cut transformation* $t_!$ *to* C.

6.1.3 The incremental Cut-and-Once transformation $t_{!+o}$

A typical top-down ILP algorithm specializes a clause by adding a literal to the body of the clause being refined. The once-transformation t_o works by transforming the entire specializ*ed* clause. Like we mentioned before, one problem with the once-transformation is that the transformed clause, which may require further specialization, is "discarded" after its coverage on the training examples is computed; if a literal is added to the body of this (transformed) clause, the once-transformation will then be applied to the newly specialized clause only to repeat any work done earlier. One can, therefore, save some work by adding a literal "directly" to the body of a (transformed) clause to make a new transformed specialized clause that is equivalent to the transformed clause produced by applying the once-transformation on the entire specialized clause. Again, the body of a transformed clause is a set of independent goals separated by the cut operator and a sequence of literals inside a $once/1$ forms an independent goal on their own. More precisely, a transformed clause can be represented as $Head \leftarrow S_1, !, \ldots, !, S_n$ (where $n \geq 0$). When $n = 0$, the body of the clause is empty. Each S_i is a conjunction of dependent literals and S_i is independent of S_j if $i \neq j$. Furthermore, each S_i contains a finite number of sequences of literals encapsulated inside a $once/1$ denoted as G_{ij} which is the jth such sequence of literals in S_i. There are totally six cases that $t_{!+o}$ needs to consider when adding a literal to a transformed clause. They are as follows:

Case 1: Merging two or more independent goals.

When a literal is to be added to the body of a transformed clause (to make new transformed specialized clause), one needs to discover new dependencies that now exist among some of the independent goals because of the addition of the literal. For example, suppose we have a transformed clause $C : h(A) \leftarrow f(A, Z), !, p(A, X)$ and a literal $L = q(X, Z)$ to be added to C (in order to further specialize it). There are two independent goals in the body of C: $f(A, Z)$, and $p(A, X)$. They are independent of each other because they don't share any non-ground variable. Adding L to the body of C will create new dependency among them because $f(A,Z)$ share a non-ground variable with L (the variable Z) and similarly $p(A,X)$ share a non-ground variable with L (X), making all three of them dependent on each other. Thus the result of adding L to C is the clause $C' \leftarrow f(A,Z), p(A,X), q(X,Z)$ where the two independent goals have "merged" with L to form a new goal with three literals.

Case 2: Extending a set of literals encapsulated inside a once/1 in an independent goal.

One also needs to extend some existing dependencies among a set of literals within an independent goal to include the new literal. For example, we've a transformed clause $C : t(A) \leftarrow p(A,S), !, s(A,X,Y), once(p(Y,U), q(U,W), k(W,T1)), once(d(X,V), e(V))$ and a literal $L = l(W,T2)$ to be added to the body of C for a specialization of C. There are two independent goals, S_1 and S_2, in the body of C: $S_1 = p(A,S)$, and $S_2 = s(A,X,Y), once(p(Y,U), q(U,W), k(W,T1)), once(d(X,V), e(V))$. Adding L to the body of C means adding L to either S_1 or S_2. Since L shares a non-ground variable with S_2 (W), there exists dependency between S_2 and L. Therefore, L should be added to S_2. There are two separate groups of literals, G_{21} and G_{22}, inside a $once/1$: $G_{21} = once(p(Y,U), q(U,W), k(W,T1))$, and $G_{22} = once(d(X,V), e(V))$. G_{21} shares a non-ground variable with with L (W) and, hence, L should be added to G_{21}, which results in an "extended" set of literals encapsulated inside a $once/1$: $once(p(Y,U), q(U,W), k(W,T1), l(W,T2))$. The result of adding L to C is $C' : t(A) \leftarrow p(A,S), !, s(A,X,Y), once(p(Y,U), q(U,W), k(W,T1), l(W,T2)), once(d(X,V), e(V))$.

Case 3: Merging independent goals and extending a set of literals encapsulated inside a once/1 in an independent goal.

Sometimes, one has to handle case 1) and 2) at the same time; if the literal to be added to the transformed clause shares a non-ground variable with a literal in an independent goal and one of the literals inside a $once/1$ in a different independent goal. For example, we have the transformed clause $C = t(A) \leftarrow p(A,S), !, s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1))), once((d(X,V), e(V))$ and $L = l(T1,S)$. There are two independent goals, S_1 and S_2, in C: $S_1 = p(A,S)$ and $S_2 = s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1))), once((d(X,V), e(V))$.

There are two separate groups of literals inside S_2, G_{21} and G_{22}, encapsulated by $once/1$: $G_{22} = once(d(X,V), e(V))$, and $G_{21} = once(p(Y,U), q(U,W), k(W,T1))$. L shares a non-ground variable with S_1 and G_{21}, which are S and $T1$ respectively. Now, there exists a new dependency between S_1 and S_2 through the literal L (which shares a non-ground variable with each of them). Since there are now new dependencies among the literals $p(A,S)$, $l(T1,S)$, and all the literals in G_{21}, G_{21} needs to be extended to include $p(A,S)$, and $l(T1,S)$. Therefore, the result of adding L to C is $C' = t(A) \leftarrow s(A,X,Y), once((d(X,V), e(V))), once((p(A,S), p(Y,U), q(U,W), k(W,T1), l(T1,S)))$.

Case 4: Merging independent goals and forming a new once/1 to encapsulate a set of dependent literals.

One also needs to merge independent goals and create a new $once/1$ to encapsulate a set of literals for which new dependencies exist among them through sharing non-ground variables with the new literal to be added to the transformed clause. Suppose we have $C = t(A) \leftarrow p(A,S), !, s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1)))$, $once((d(X,V), e(V))$, and $L = l(X,S)$. Again, there are two independent goals S_1 and S_2 as we mentioned. This time, L shares a non-ground variable, S, with $p(A,S)$ in S_1 and the non-ground variable, X, with $G_{22} = once((d(X,V), e(V)))$ and $s(A,X,Y)$ in S_2. Since new dependencies exist among $p(A,S)$, $s(A,X,Y)$, and $once((d(X,V), e(V)))$ through their sharing non-ground variables with $l(X,S)$, they have to be "merged" together somehow in a way that is free of unnecessary backtracking on any one of these goals. There are two possible (equivalent) solutions and one of them requires creating a new $once/1$ to encapsulate a set of dependent literals: $C' = t(A) \leftarrow s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1))), once((d(X,V), e(V)))$, $p(A,S), l(X,S)$, and $C' = t(A) \leftarrow s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1)))$, $once((p(A,S), l(X,S))), once((d(X,V), e(V)))$.

This clause allows unnecessary backtracking on the goals $p(A,S)$ and $l(X,S)$: $C' = t(A) \leftarrow s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1))), p(A,S), l(X,S), once((d(X,V)$, $e(V)))$. If one always encapsulates literals that should be inside a $once/1$, then one doesn't have to deal with computing the correct ordering of these independent goals.

Case 5: Merging independent goals but splitting a set of literals encapsulated by a once/1 and forming a new once/1 to group a set of dependent literals.

Sometimes, one needs to take out a subsequence of literals inside a $once/1$ to allow backtracking on these subsequence of literals for generating alternative solutions. Suppose we have $C : t(A) \leftarrow p(A,S), !, s(A,X,Y), once((p(Y,U), q(U,W), k(W,T1)))$, $once((d(X,V), e(V))$ and $L = l(U,S)$. Again, there are two independent goals S_1 and

S_2 as we mentioned. This time, L shares a non-ground variable, S, with $p(A,S)$ in S_1 and the non-ground variable, U, with $G_{21} = \mathit{once}((p(Y,U), q(U,W), k(W,T1)))$. Therefore, $p(A,S)$, $l(U,S)$, and $G_{21} = \mathit{once}((p(Y,U), q(U,W), k(W,T1)))$ have to be somehow merged together because of new dependencies that exist among them. Next, for each G_{ij} which shares a non-ground variable with L, find the first literal J in G_{ij} such that 1) J shares a non-ground variable with L, and 2) there is a non-ground variable in J which appears in the sequence of literals containing the literal next to J (counting from left to right) up to the last literal in G_{ij}. Let G'_{ij} be the sequence of literals containing the literal J up to the last literal in G_{ij} (counting from left to right starting at J). Let G''_{ij} be the sequence of literals such that $G_{ij} = G''_{ij} \cup G'_{ij}$ (i.e. G''_{ij} is the sequence of literals in G_{ij} that comes before G'_{ij}). Then, encapsulate the sequence of literals G'_{ij} inside a $\mathit{once}/1$. Let's call it a sequence of "frozen literals". The sequence of literals G''_{ij} will not be encapsulated by a $\mathit{once}/1$. Let's call it a sequence of "free literals". Then, form a new sequence of literals S by "concatenating" G'_{ij} for all i and all j. Next, make a new independent goal $\mathit{once}(S_1 \cup \{L\})$. Finally, put all free literals (in any order) before all frozen literals to make the new body of the transformed clause. In out example, G_{21} shares a non-ground variable U with $l(U,S)$, and the first literal in G_{21} that satisfies conditions 1) and 2) mentioned is $q(U,W)$ since 1) $q(U,W)$ shares the non-ground variable U with $l(U,S)$, and 2) the non-ground variable W in $q(U,W)$ appears in the sequence of literals $k(W,T1)$ (which has only one literal in this particular case). So, we have $G'_{21} = q(U,W), k(W,T1)$ and $G''_{21} = p(Y,U)$. We also need to form a new independent goal $\mathit{once}((p(A,S), l(U,S)))$. $C' = t(A) \leftarrow$

$s(A, X, Y), p(Y, U), once((d(X, V), e(V))), once((q(U, W), k(W, T1))),$ $once((p(A, S), l(U, S)))$ is the clause resulted from adding L to C.

Case 6: Creating a new independent goal.

If the literal to be added to the body of the transformed clause shares no non-ground variables with the body of the transformed clause, then one just simply needs to create a new independent goal containing the literal. For example, we have $C = t(A) \leftarrow p(A, S), !, s(A, X, Y), once((p(Y, U), q(U, W), k(W, T1))), once((d(X, V), e(V))$, and $L = g(A, Q)$. Obviously, L shares no non-ground variables with any of the independent goals in the body of C and we create a new independent goal $g(A, C)$ in the body of C. $C' = t(A) \leftarrow p(A, S), !, s(A, X, Y), once((p(Y, U), q(U, W), k(W, T1))),$ $once((d(X, V), e(V)), !, g(A, Q)$ is the resulted clause.

6.2 Query Transformation Complexity

We will use the following symbols in analysing the complexity of transforming a clause: v, the maximum number of variables in a literal; and N, the number of literals in the clause being transformed.

6.2.1 Complexity of $t_!$

Theorem 3 (Costa et al., 2002). The complexity of $t_!$ is $\mathcal{O}(vN^2)$.

Proof. The dominant component of $t_!$ stems from calculating groups of dependent literals. The algorithm starts with an empty set of groups. In the i-th iteration it creates a new group containing the i-th literal in the clause, $\{l_i\}$, and merges this group with all existing groups that share variables with l_i. Testing if a group j shares variables with l_i is $\mathcal{O}(vn_j)$ where n_j is the number of literals in the group. This linear intersection test is possible because the algorithm keeps a sorted list of variables for each group. Performing the intersection test on all groups is $\mathcal{O}(vi)$ because $\sum_j n_j = i - 1$. Now assume g_i groups have been identified that share with $\{l_i\}$; these have to be merged. Merging one group is $\mathcal{O}(vi)$, so merging g_i groups is $\mathcal{O}(g_i \cdot vi)$. The total complexity of $t_!$ is $T(N) = \mathcal{O}\left(\sum_{i=1}^{N} vi + g_i \cdot vi\right) = \mathcal{O}\left(vN^2 + vN \cdot \sum_{i=1}^{N} g_i\right)$. Since $\sum_{i=1}^{N} g_i < N$, this is just $\mathcal{O}(vN^2)$. $\square$

6.2.2 Complexity of t_o

Theorem 4 (Costa et al., 2002). The complexity of t_o is $\mathcal{O}(vN^3)$.

Proof. The implementation of t_o first finds independent groups of literals using the same algorithm as the cut-transformation. If all literals are in the same group then the algorithm grounds the variables of the first literal and calls itself recursively for the other literals. If the literals are partitioned into several groups then the algorithm does a recursive call for each group. In the first case, the time complexity is $T(N) = \mathcal{O}(vN^2) + T(N-1)$ where v is the maximum number of variables in a literal and N the number of literals in the clause. If each recursive call is of this type, then the total complexity is $\mathcal{O}(vN^3)$. In the second case $T(N) = \mathcal{O}(vN^2) + \sum_{i=1}^{g_N} T(n_i)$ with $g_N > 1$ the number of subgroups and n_i the number of literals in subgroup i. It can be shown that the worst case is $g_N = 2$, $n_1 = 1$ and $n_2 = N - 1$.[4] The overall complexity is $\mathcal{O}(vN^3)$ for both cases. $\square$

[4]To see this, note that $T(N-i) + T(i) < T(N-1) + T(1) < T(N)$ when T is a superlinear function and $1 < i < N$. In our case T has to be superlinear because it contains an N^2 term. The demonstrandum follows by recursively applying these inequalities on the terms in $\sum_i T(n_i)$.

6.2.3 Complexity of $t_{!+o}$

Theorem 5. Given a transformed clause with N literals in the body, v the maximum number of variables in a literal, and n the number of independent groups of literals S_i in the body of the transformed clause, G_{ij} is the jth sequence of literals encapsulated inside a *once*/1 in S_i, L is the literal to be added to the body of the transformed clause, the worst case complexity of $t_{!+o}$ is $\mathcal{O}(nvN^2)$.

Proof. The complexity of checking if a literal or an S_i in the body of the transformed clause shares a non-ground variable with L is $\mathcal{O}(v)$. There are different cases that $t_{!+o}$ needs to consider. Some has a lower complexity than others. For example, in case 1 (merging two or more independent goals) and case 6 (creating a new independent goal), the complexity of $t_{!+o}$ is only $\mathcal{O}(vn)$. However, we are going to present the worst case complexity of $t_{!+o}$ here which is case 5 (merging independent goals but splitting a set of literals encapsulated by a once/1 and forming a new once/1 to group a set of dependent literals). The time it takes to find the first literal J in G_{ij} that satisfies the two conditions 1) and 2) mentioned is $\mathcal{O}(vN^2)$ since, in the worst case, we might have N literals in G_{ij} and one needs to check for each literal in G_{ij} (from left to right and starting with the first literal in G_{ij}) if there is a non-ground variable in it which appears in the rest of G_{ij}. The complexity of checking, for a literal in G_{ij}, if there is a non-ground variable in it which appears in the rest of G_{ij} is $\mathcal{O}(vN)$. Therefore, the total complexity of doing this for all the literals in G_{ij} is $\mathcal{O}(vN^2)$. Since we have at most n such G_{ij}'s (that shares a non-ground variable with L), the worst case complexity of $t_{!+o}$ is $\mathcal{O}(nvN^2)$. □

We will discuss the significance of these theoretical results in Section 6.4.

6.3 Query Execution Complexity

We now examine the complexity arising from executing a transformed query with a view to estimating the gain in efficiency obtained from performing a transformation. We further consider estimates of average gains only, as some of the transformations cannot always guarantee efficiency gain. We will use the following notation:

- q represents a query with literals $l_1, l_2, \cdots, l_N$
- N is the number of literals in the query (as before)
- e is the number of examples
- b is the average branching factor of the SLD tree of the query and can be seen as a measure of non-determinacy of the data
- d is the depth of the SLD tree
- q_i and m are defined by $t_!(q) = q_1, !, q_2, !, \ldots, q_m, !$; we call the q_i independent subgoals
- d_i is the depth of the SLD tree generated by q_i
- N_i is the number of literals in q_i

6.3.1 Complexity of $t_!$

We approach the efficiency gain yielded by $t_!$ from two points of view: first, looking at SLD-trees and non-determinacy; second, at a higher abstraction level, looking at the execution times the subgoals consume.

Let us assume for now that a literal l_i succeeds $k_i > 0$ times. Then without the transformation the number of nodes in the SLD tree is $\prod_i k_i$. After the transformation it becomes $\sum_i \prod_{l_j \in q_i} k_j$. If we simplify this by assuming a constant branching factor b, execution of the original query takes time $\mathcal{O}(b^N)$ while execution of the transformed query takes time $\mathcal{O}(b^{\max N_i})$. Thus, the execution time of the query is reduced from "exponential in its length" to "exponential in the length of the longest conjunction between two cuts".

We can obtain some more insights from a second stance, describing efficiency in terms of times rather than SLD tree sizes. We introduce

- t_i : the time needed for exhausting the search space of q_i
- s_i : the number of times q_i succeeds
- $\bar{t_i} = t_i/(s_i + 1)$: the average time until success or failure for subgoal q_i
- $k = \min\{i|s_i = 0\}$ (i.e. q_k is the first subgoal that has no solutions)

Without cuts the time needed to confirm failure of the clause on a single example is

$$t_1 + (s_1(t_2 + s_2(....))) = t_1 + s_1t_2 + ... + s_1...s_{k-1}t_k$$

and after applying the cut-transformation this becomes

$$\bar{t}_1 + \bar{t}_2 + ... + \bar{t}_{k-1} + t_k$$

If the last term in both formulae is dominant (which happens if t_k is large), a speedup of at most $s_1s_2 \cdots s_{k-1}$ can be obtained. If an earlier t_j dominates, this results in a speedup of roughly $s_1s_2 \cdots s_{j-1}$.

6.3.2 Complexity of t_o

The once-transformation, t_o, is the recursive application of $t_!$, and thus has a similar effect on efficiency; only now the original equivalence classes may be subdivided further.

The execution time of the transformed query is then exponential in the length of the longest conjunction between two cuts minus the length of the longest conjunction within a `once` construct within that longest conjunction between two cuts. More precisely, if b is the branching factor, the execution time it takes to execute a query after applying t_o is

$$\mathcal{O}(b^{max_i N_i - max_j M_{ij}}).$$

N_i is the length of the ith conjunction between two cuts, and M_{ij} is the length of the jth conjunction inside a `once` construct inside the ith conjunction within two cuts.

6.3.3 Complexity of $t_{!+o}$

The transformed clause produced by t_o is equivalent to the transformed clause produced by $t_{!+o}$ given the transformed clause (without the newly added literal) and the new literal to be added. More precisely, suppose $C : H \leftarrow B$ and $C1 : H \leftarrow B \cup \{L\}$ where L is a literal. We have $t_o(C1) = t_{!+o}(t_o(C), L)$. Therefore, the query produced by the incremental cut-and-once transformation has the same execution complexity as that of the once transformation, although they have different transformation complexities.

6.4 Conclusion

The results on computational complexity can be summarized as follows:

- As the transformation time itself does not depend on the size of the data set, for large data sets it will become negligible compared to the execution time gained by performing the transformation.
- Concerning the evaluation of clauses, the efficiency gain the transformations yield depends on the size of the SLD tree of the original and transformed clause. This size is affected by:
  - the non-determinacy of the literals in the clause, which measures the branching factor in the SLD tree
  - the number of examples, which determines the number of head variable substitutions for the clauses
  - the depth of the SLD tree, which can be considered proportional to the length of the clause for the untransformed query; for the transformed query it is proportional to the length of the longest conjunction inside a `once` construct in the body of the transformed clause.

This yields four parameters that influence efficiency gain: non-determinacy, number of examples, length of clauses, and the complexity of the most complex subgoal (i.e. the longest conjunction inside a `once` construct in the body of the transformed clause). A maximal gain can be expected when the first three parameters have large values and the last one is small.

Although the execution time of a transformed query is still (theoretically) exponential in complexity. Practically, the gain in performance is significant because we are reducing the execution complexity from *large* exponential to *small* exponential. Readers can refer to (Costa et al., 2002) for details on experimental results on the gain in execution time of a transformed query.

The complexity of applying t_o is $\mathcal{O}(vN^3)$ while that of $t_{!+o}$ is $\mathcal{O}(nvN^2)$. Since the number of independent groups of literals (i.e. S_i) cannot be more than the number of literals in the body of a clause, $n \leq N$. In practice, $n = N$ is a very (rare) extreme case, which happens only when each literal in the body of a clause is independent of all other literals. Theoretically speaking, the transformation complexity of $t_{!+o}$ is a slight improvement over that of t_o. The practical improvement in CPU time is not going to be obvious since the respective complexities of t_o and $t_{!+o}$ are still too close. It is even less of a concern given the performance of the machines available nowadays, except, perhaps, if one has to learn a theory with a lot of clauses in which each clause has a lot of literals in the body.

In conclusion, we have presented three transformation algorithms: $t_!$, t_o, and $t_{!+o}$. $t_!$ does not take into account more refined details of dependency amoung literals inside a conjunction between two cuts while the other two transformation do. The transformation complexity of $t_{!+o}$ is only theoretically better than that of t_o while they have the same query execution complexities.

Chapter 7

Related Work

There are two major approaches in ILP. First, in a top-down ILP approach, one searches the hypothesis space from general to specific. Second, in a bottom-up ILP approach, one searches the hypothesis space from specific to general. Combining both approaches has been proven to produce ILP systems that are better than traditional ILP systems from each approach alone in many different domains. Integrated ILP systems have been developed for two tasks: learning semantic parsers given a set of natural language sentences using CHILLIN (Zelle & Mooney, 1994), and mining relational data represented as PROLOG facts using ALEPH (Zelezny et al., 2002). Each of these earlier integrated ILP systems has its own shortcomings.

7.1 The Strength and Weakness of CHILLIN

The task of inducing semantic parsers requires learning two type of features of a parse state for disambiguation: its functional structure and contextual information. CHILLIN, an ILP algorithm applied on this task, combines the two traditional

ILP approaches "sequentially" by first constructing the least general generalization (LGG) of a random pair of examples and then specializing this generalization in a FOIL-like manner. A weakness of traditional top-down ILP algorithm, e.g. FOIL, is that it cannot learn clauses with function terms. Therefore, it is not suitable for capturing functional structures of parse states. A strength of CHILLIN is that it can learn clauses with function terms (by constructing an initial generalization of a random pair of positive examples which is their LGG in the first step of learning a clause), which makes it possible to capture functional structures in parse states. However, a weakness of CHILLIN is that important contextual information, e.g. the presence of a certain word in the input buffer of a parse state, can be lost in the process of computing the LGG of a random pair of positive examples, making it difficult to learn the context of a parse state for disambiguation.

7.2 Overcoming Limitations of CHILLIN in COCKTAIL

The functional structure of a parse state can be more effectively learned by a bottom-up ILP approach while the context of a parse state can be more effectively learned by a top-down ILP approach. The functional structure of a parse state can be more effectively learned by a bottom-up ILP approach because the LGG of a pair of positive examples with function terms produces a generalization of these positive examples as well as a generalization of the function terms embedded in them. The context of a parse state can be more effectively learned by a top-down ILP approach through the learning of literals with constants that appear in a parse state (a.k.a. "theory constants" (Quinlan, 1996)). One can learn both parsing features "individually" by applying the most suitable ILP approach for learning each type of parsing features. Cocktail (Tang & Mooney, 2001) is a new ILP algorithm which combines 1) the LGG operator in Golem (Muggleton & Feng, 1990), and 2) m-Foil (Lavrac & Dzeroski, 1994b) in a coherent mechanism for learning both types of features in a parse state. Different combinations of parsing features learned produce theories of varying sizes. Cocktail chooses the best combination of parsing features by selecting the theory with the smallest size, which is a heuristic based on the principle of minimum description length (Rissanen, 1978). Experimental results on learning semantic parsers for two databases demonstrated that Cocktail performed better than the previous best method for this task which is Chillin.

7.3 The Strength and Weakness of m-Foil

A strength of a top-down approach like m-Foil is that the generation of literals is directed by the heuristic search process itself: only the set of literals that make refinements to clauses in the search beam are generated. Clauses with insufficient heuristic value are discarded, saving the need to generate literals for them.

However, a major weakness of the top-down approach is that the enumeration of all possible combination of variables generates many more literals than necessary; some literals generated by the algorithm are not even guaranteed to cover one positive example, which wastes CPU time in testing them. Even worse, the complexity of enumerating all such combinations is exponential w.r.t. the arity of the predicates (Pazzani & Kibler, 1992). The complexity issue did not show up in our experimental results outlined in Section 5.5.3 because the average arity of the background predicates in our domain was only two. In domains with a significantly higher average arity of the background predicates, a traditional top-down approach will become very inefficient in learning.

7.4 The Strength and Weakness of Aleph

The Aleph ILP system is based on a theory in ILP called inverse entailment (Muggleton, 1995) which learns a hypothesis by first constructing the most specific hypothesis covering a seed example (a.k.a bottom clause) and then looking for a generalization of the bottom clause that produces a good value from the search heuristic. A strength of such an approach is that a literal is created using a ground atom describing a known positive example. The advantages are: 1) specializing a clause using this literal results in a clause that is guaranteed to cover at least the seed example, and 2) the set of literals generated are constrained to those that satisfy 1), which substantially reduces the complexity of generating literals.

The size of the subsumption lattice, i.e. the hypothesis space, bounded by the bottom clause is two raised to the power of the size of the bottom clause. A problem with Aleph is that the size of the bottom clause constructed by it is exponential in complexity (Muggleton, 1995). When mining relational data with a large number of background facts as in link discovery, the bottom clause constructed by Aleph becomes very large, making the size of the hypothesis space intractably large. Learning becomes very inefficient as one has to search such a huge space of hypotheses.

7.5 Overcoming Limitations of Aleph and m-Foil in Beth

Beth is a new integrated ILP algorithm for relational data mining. By using a top-down approach to heuristically guide the construction of generalizations of a bottom clause, Beth combines the strength of both approaches. More precisely, we no longer build the bottom clause using a random seed example before we start searching for a good clause. Instead, after a seed example is chosen, one generates literals in a top-down fashion, i.e. guided by heuristic search, except that the literals generated are constrained to those that cover the seed example. Learning patterns for detecting potential terrorist activity is a current challenge problem for relational data mining. Experimental results on artificial data for this task with over half a million facts show that Beth is significantly more efficient at discovering such patterns than two leading ILP systems: Aleph and m-Foil.

Chapter 8

Future Work

8.1 Caching Ground Atoms in Beth

One shortcoming of an approach like Beth is that searching a smaller hypothesis space comes with a cost of spending more time on constructing each hypothesis. This is primarily because of the need to find ground atoms satisfying all refinement constraints during the construction of a hypothesis.

In the entire course of learning a theory, the same ground atom satisfying all the refinement constraints can be used in making different literals to different clauses in the search beam. For example, suppose we have two clauses in the current search beam

$$C1 = t(A) \leftarrow f(A, B)$$
$$C2 = t(A) \leftarrow h(A, B)$$

where $t(a) \leftarrow f(a, b)$ and $t(a) \leftarrow h(a, c)$ are the cached proofs of $C1$ and $C2$ respectively. Obviously, at this point, the current building bottom clause is $t(a) \leftarrow$

$f(a,b), h(a,c)$. Further suppose that both $C1$ and $C2$ are still covering negative examples. We don't care how $C1$ is going to be specialized. However, we do know that one obvious way to specialize $C2$ is by using the ground atom $f(a,b)$ to make the literal $f(A,C)$. Thus, the ground atom $f(a,b)$ is used in making the literal $f(A,B)$ for $C1$ and the literal $f(A,C)$ for $C2$, although at different times. Therefore, it might be possible that caching ground atoms that satisfy all refinement constraints in the course of making literals can substantially improve the rate of constructing and testing hypotheses as the same ground atom needs only to be fetched in the cache instead of from the entire set of background facts. The body of the building bottom clause can serve exactly this purpose because it contains exactly the set of ground atoms satisfying all refinement constraints when making literals.

The actual gain in performance, if any, is going to depend on the actual difference between the "table look-up" time given the entire set of background facts and that of given only those ground atoms in the body of the building bottom clause. It remains an open question whether it is possible to find an approach such that:

1. it dynamically and selectively searches a smaller but relevant portion of the hypothesis space

2. it maintains a high rate of constructing and testing hypotheses

8.2 Boosting Beth Using Decorate

Ensemble methods like bagging and boosting that combine the decisions of multiple hypotheses are some of the strongest existing machine learning methods. The *diversity* of the members of an ensemble is known to be an important factor in determining its generalization error. DECORATE is a new method for generating ensembles that directly constructs diverse hypotheses using additional artificially-constructed training examples (Melville & Mooney, 2003). The technique is a simple, general meta-learner that can use any strong learner as a base classifier to build diverse committees. Experimental results using decision-tree induction as a base learner demonstrate that this approach consistently achieves higher predictive accuracy than both the base classifier and bagging (whereas boosting can occasionally decrease accuracy), and also obtains higher accuracy than boosting early in the learning curve when training data is limited. A future work is to use BETH as a base learner in DECORATE to learn a diversified set of hypotheses for achieving even better predictive accuracies in existing or new problems in the EELD program or other ILP problems in a different domain.

8.3 Applying BETH to Other ILP Problems

The problem we attempted was classifying if a murder event is "murder-for-hire". This problem has an average arity of two; the average arity of a background predicate in the set of background knowledge is two. Another characteristic of this problem is that it has a lot of background facts. In fact, it had around 568k facts in the background knowledge, which is an unprecedented amount of facts in an ILP problem. BETH was demonstrated to outperform leading ILP methods significantly in terms of computational efficiency in such a problem.

Categorizing murder events is only one particular challenge problem in the EELD program. Another problem that has been previously attempted was classifying if two different events in a nuclear smuggling incident are linked (Mooney, Melville, Tang, Shavlik, de Castro Dutra, Page, & Costa, 2002). This problem has a much higher average arity than two since there were lots of predicates in the background knowledge with an arity as high as eleven such as the following predicate:

```
lk_person_person(person_person_id_3,
                 person_id_18,
                 person_id_20,
                 event_id_22,
                 relation_id_12,
                 ''Politov recruited Yevseev to steal the uranium'',
                 motive_number_3,
                 '?','?','?','?').
```

This predicate describes links between two people associated, 'person_id_18' and 'person_id_20', where each distinct person is identified by a unique ID number; the type of association between them is denoted by the "relation number" (e.g. relation_id_12 denotes the relation type "Recruiter"). There are other arguments in the predicates like a memo describing the event in which the people listed were involved, but we'll not dive into all the details here. It would be interesting to see how well BETH performs in problems with lots of these high-arity ground literals in the background knowledge. Problems like carcinogenesis in bioinformatics have a much smaller amount of background facts (around 24k) than a problem like categorizing murder events in link discovery. BETH is designed with handling lots of background facts in mind, it is, however, still interesting to see if and/or how much it produces gain in computational efficiency in problems with less background facts than that of link discovery.

Chapter 9

Conclusion

Two major approahces have been developed in the past decade of research in Inductive Logic Programming (ILP): 1) Top-down which searches the hypothesis space from general to specific, and 2) Bottom-up which searches the hypothesis space from specific to general. Integrating both approaches have been demonstrated to produce ILP systems that are more effective in many different domains. Integrated ILP systems have been developed for two tasks: learning semantic parsers given a set of natural language sentences using CHILLIN (Zelle & Mooney, 1994), and mining relational data represented as PROLOG facts using PROGOL (Muggleton, 1995).

The task of inducing semantic parsers requires learning two type of features of a parse state for disambiguation: its functional structure and contextual information. A bottom-up ILP approach is more effective for learning the former while a top-down approach for the latter. CHILLIN combined both ILP approaches "sequentially" in the sense that it applies a bottom-up approach to construct an initial generalization of examples as the first step in learning a hypothesis followed by a top-down approach which specializes this generalization in a FOIL-like manner. While CHILLIN was able to learn functional structures of a parse state, largely in its first step of applying bottom-up generalization, it was lacking with respect to learning contextual information like the presence or absence of a certain word or phrase in the input buffer of a parse state. COCKTAIL, a new ILP algorithm, overcomes the shortcomings of CHILLIN by using both ILP approaches in learning "individually" (in the sense that each approach is applied independent of the other) and choosing the best combination of parsing features learned from each ILP approach. Experimental results on learning semantic parsers for two databases demonstrated that COCKTAIL performed better than the previous best method for this task CHILLIN.

Relational data mining involves learning relational knowledge embedded in a huge amount of data in a relational database. The PROGOL and ALEPH ILP systems are based on a theory in ILP called inverse entailment which learns a hypothesis by first constructing the most specific hypothesis covering a seed example (a.k.a bottom clause) and then looking for a generalization of the bottom clause that produces a good value from the search heuristic. When mining relational data with a large number of background facts, the hypothesis space bounded by the bottom clause becomes intractably large, making learning very inefficient. BETH is a new integrated ILP algorithm for relational data mining. A top-down approach heuristically guides the construction of clauses without building a bottom clause; however, it wastes time exploring clauses that cover no positive examples. By using a top-down approach to heuristically guide the construction of generalizations of a bottom clause, BETH combines the strength of both approaches. Learning patterns for detecting potential terrorist activity is a current challenge problem for relational data mining. Experimental results on artificial data for this task with over half a million facts show that BETH is significantly more efficient at discovering such patterns than two leading ILP systems: ALEPH and M-FOIL.

In conclusion, we have presented two more advanced integrated ILP systems than earlier ILP hybrids like CHILLIN and ALEPH: COCKTAIL and BETH. The former was applied to the task of learning semantic parsers, which produced better accuracy in parsing than the previous best approach CHILLIN. The latter was applied to mining relational data for detecting potential terrorist activity, which was shown to give better efficiency in mining such patterns than the previous best approach ALEPH as well as another leading top-down approach M-FOIL. Both ILP systems were demonstrated to perform better than previous leading ILP systems in each domain.

Appendix A

Screenshots of A Natural Language Interface

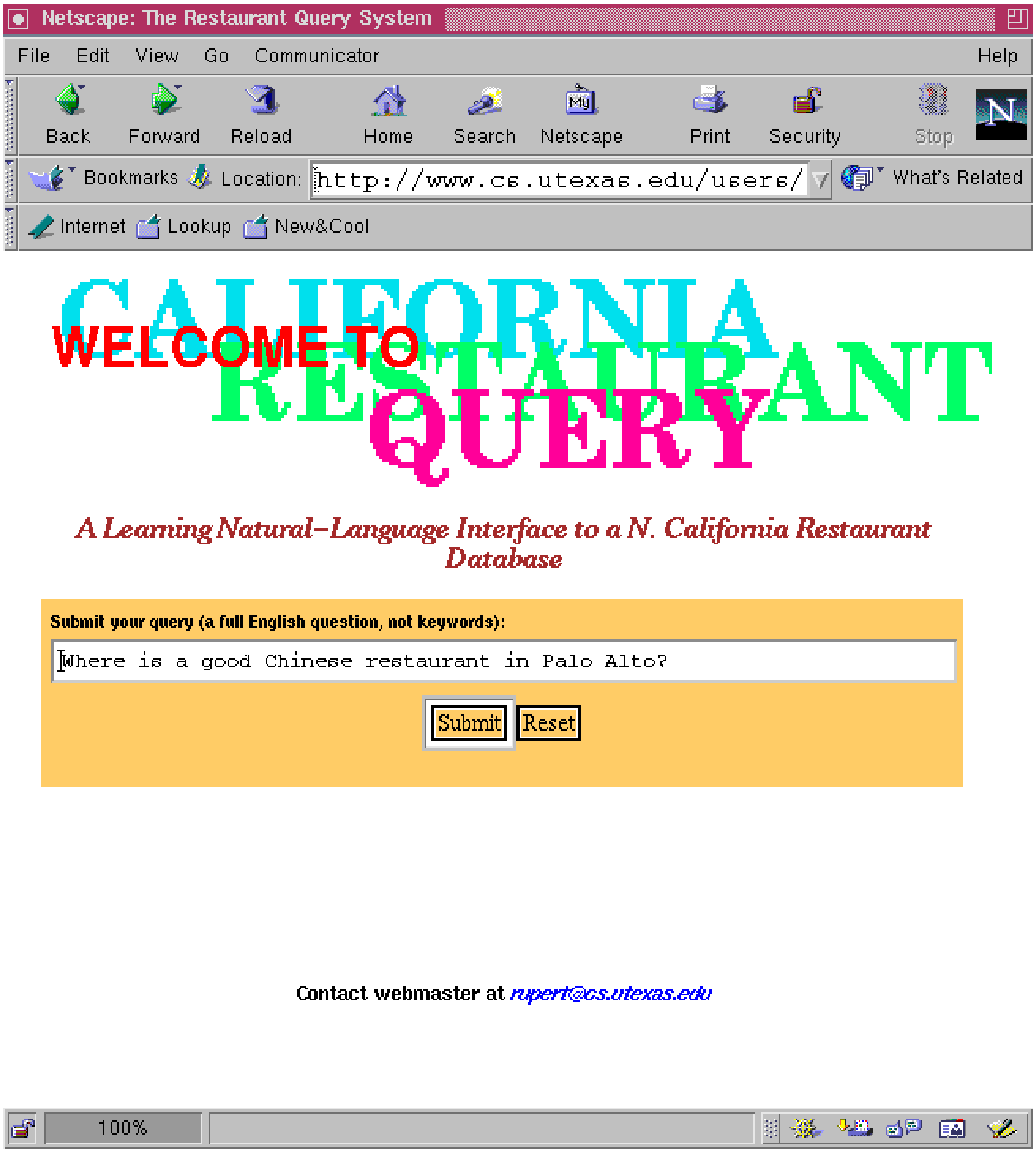
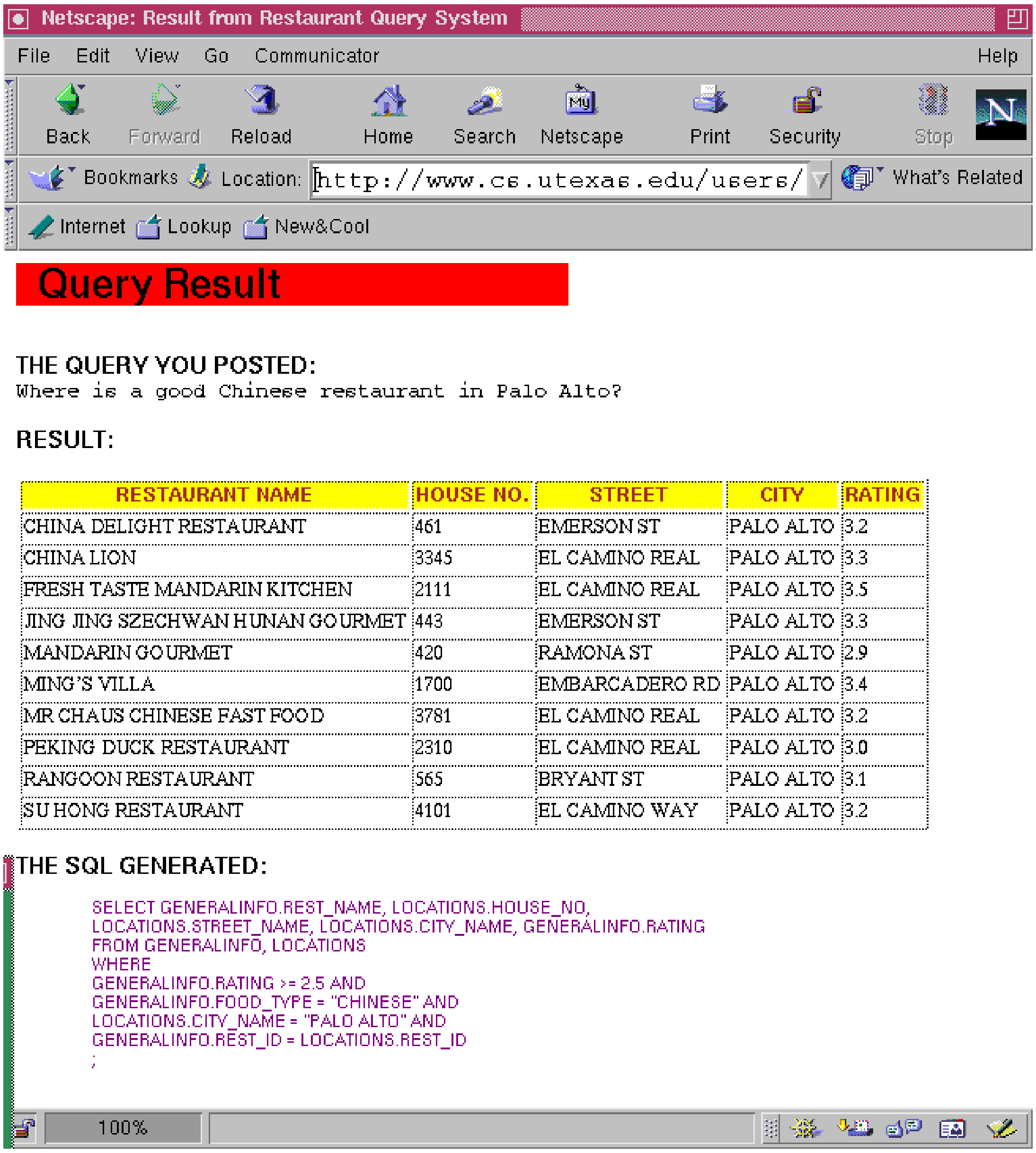


Figure A.1: A screenshot of a user posting a question to our learned Web-based NL Database Interface



Query Result

THE QUERY YOU POSTED:

Where is a good Chinese restaurant in Palo Alto?

RESULT:

RESTAURANT NAME	HOUSE NO.	STREET	CITY	RATING
CHINA DELIGHT RESTAURANT	461	EMERSON ST	PALO ALTO	3.2
CHINA LION	3345	EL CAMINO REAL	PALO ALTO	3.3
FRESH TASTE MANDARIN KITCHEN	2111	EL CAMINO REAL	PALO ALTO	3.5
JING JING SZECHWAN HUNAN GOURMET	443	EMERSON ST	PALO ALTO	3.3
MANDARIN GOURMET	420	RAMONA ST	PALO ALTO	2.9
MING'S VILLA	1700	EMBARCADERO RD	PALO ALTO	3.4
MR CHAUS CHINESE FAST FOOD	3781	EL CAMINO REAL	PALO ALTO	3.2
PEKING DUCK RESTAURANT	2310	EL CAMINO REAL	PALO ALTO	3.0
RANGOON RESTAURANT	565	BRYANT ST	PALO ALTO	3.1
SU HONG RESTAURANT	4101	EL CAMINO WAY	PALO ALTO	3.2

THE SQL GENERATED:

```
SELECT GENERALINFO.REST_NAME, LOCATIONS.HOUSE_NO,
LOCATIONS.STREET_NAME, LOCATIONS.CITY_NAME, GENERALINFO.RATING
FROM GENERALINFO, LOCATIONS
WHERE
GENERALINFO.RATING >= 2.5 AND
GENERALINFO.FOOD_TYPE = "CHINESE" AND
LOCATIONS.CITY_NAME = "PALO ALTO" AND
GENERALINFO.REST_ID = LOCATIONS.REST_ID
;
```


Figure A.2: A screenshot of answers generated for a user's question by our learned Web-based NL Database Interface

Appendix B

The U.S. Geography Corpus

A sample of 880 sentences and associated queries expressed in Prolog of the U.S. Geography corpus are shown here. The sentences are printed out as unquoted Prolog literals. So, there is no capitalization and the final punctuation is separated from the last word of the sentence. The full set of data can be downloaded on the Web via `ftp://ftp.cs.utexas.edu/pub/mooney/nl-ilp-data/geosystem/geoqueries880`. Here is a sample set of sentences from the U.S. Geography corpus:

```
Sentence:
give me the cities in virginia .
Logical Query:
answer(A, (city(A),loc(A,B),const(B,stateid(virginia)))).


Sentence:
what are the high points of states surrounding mississippi ?
Logical Query:
```

```
answer(A, (high_point(B,A),loc(A,B),state(B),next_to(B,C),
           const(C,stateid(mississippi)))).
```

Sentence:

name the rivers in arkansas .

Logical Query:

```
answer(A, (river(A),loc(A,B),const(B,stateid(arkansas)))).
```

Sentence:

name all the rivers in colorado .

Logical Query:

```
answer(A, (river(A),loc(A,B),const(B,stateid(colorado)))).
```

Sentence:

can you tell me the capital of texas ?

Logical Query:

```
answer(A, (capital(A),loc(A,B),const(B,stateid(texas)))).
```

Sentence:

could you tell me what is the highest point in the state
of oregon ?

Logical Query:

```
answer(A, highest(A,(place(A),loc(A,B),state(B),
                  const(B,stateid(oregon))))).
```

Sentence:

give me all the states of usa ?

Logical Query:

```
answer(A, (state(A),loc(A,B),const(B,countryid(usa)))).
```

Sentence:

give me the cities in texas ?

Logical Query:

```
answer(A, (city(A),loc(A,B),const(B,stateid(texas)))).
```

Sentence:

give me the cities in usa ?

Logical Query:

```
answer(A, (city(A),loc(A,B),const(B,countryid(usa)))).
```

Sentence:

give me the cities in virginia ?

Logical Query:

```
answer(A, (city(A),loc(A,B),const(B,stateid(virginia)))).
```

Sentence:

give me the cities which are in texas ?

Logical Query:

```
answer(A, (city(A),loc(A,B),const(B,stateid(texas)))).
```

```
Sentence:
give me the lakes in california ?
Logical Query:
answer(A, (lake(A),loc(A,B),const(B,stateid(california)))).
```

```
Sentence:
give me the largest state ?
Logical Query:
answer(A, largest(A,state(A))).
```

```
Sentence:
give me the longest river that passes through the us ?
Logical Query:
answer(A, longest(A,(river(A),traverse(A,B),
                  const(B,countryid(usa))))).
```

```
Sentence:
give me the states that border utah ?
Logical Query:
answer(A, (state(A),next_to(A,B),const(B,stateid(utah)))).
```

```
Sentence:
```

how big is alaska ?

Logical Query:

answer(A, (size(B,A),const(B,stateid(alaska)))).

Sentence:

how big is massachusetts ?

Logical Query:

answer(A, (size(B,A),const(B,stateid(massachusetts)))).

Sentence:

how big is new mexico ?

Logical Query:

answer(A, (size(B,A),const(B,stateid(new mexico)))).

Sentence:

how big is north dakota ?

Logical Query:

answer(A, (size(B,A),const(B,stateid(north dakota)))).

Sentence:

how big is texas ?

Logical Query:

answer(A, (size(B,A),const(B,stateid(texas)))).

Sentence:

how big is the city of new york ?

Logical Query:

```
answer(A, (size(B,A),const(B,cityid(new york',C)))).
```

Sentence:

how high are the highest points of all the states ?

Logical Query:

```
answer(A, (elevation(B,A),highest(B,(place(B),loc(B,C),
                                         state(C))))).
```

Sentence:

how high is guadalupe peak ?

Logical Query:

```
answer(A, (elevation(B,A),const(B,placeid(guadalupe peak)))).
```

Sentence:

how high is mount mckinley ?

Logical Query:

```
answer(A, (elevation(B,A),const(B,placeid(mount mckinley)))).
```

Appendix C

The Job Posting Corpus

A sample of 640 sentences and associated queries expressed in Prolog of the job-posting corpus are shown here. The sentences are printed out as unquoted Prolog literals. So, there is no capitalization and the final punctuation is separated from the last word of the sentence. The full set of data can be downloaded on the Web via `ftp://ftp.cs.utexas.edu/pub/mooney/nl-ilp-data/jobsystem/jobqueries640`. Here is a sample of sentences from the corpus:

```
Sentence:
what jobs are there for web developer who know vb ?
Logical Query:
answer(A, (job(A),title(A,B),const(B,Web Developer),language(A,C),
          const(C,vb))).


Sentence:
what systems analyst jobs are there in austin ?
```

```
Logical Query:
answer(A, (title(A,W),const(W,Systems Analyst),job(A),
           loc(A,P),const(P,austin)))).


Sentence:
what jobs pay 60000 are located in austin and require a bscs ?
Logical Query:
answer(A, (job(A),salary_greater_than(A,60000,year),loc(A,C),
           const(C,austin),req_deg(A,B),const(B,BSCS)))).


Sentence:
what jobs pay 60000 are located in austin and require a degree ?
Logical Query:
answer(A, (job(A),salary_greater_than(A,60000,year),loc(A,C),
           const(C,austin),req_deg(A)))).


Sentence:
what jobs are there for pascal programers who dont know vb ?
Logical Query:
answer(A, (job(A),language(A,P),const(P,pascal),
           \+ ((language(A,C),const(C,vb)))))).


Sentence:
which jobs in austin offer for students fresh out of college
```

```
in networking ?
Logical Query:
answer(A, (job(A),loc(A,D),const(D,austin),\+req_exp(A),
           area(A,C),const(C,networking)))).

Sentence:
which jobs in houston offer over 50000 in graphics ?
Logical Query:
answer(A, (job(A),loc(A,C),const(C,houston),
           salary_greater_than(A,50000,year),
           area(A,B),const(B,graphics)))).

Sentence:
which jobs at trilogy deal with vb ?
Logical Query:
answer(A, (job(A),company(A,C),const(C,Trilogy),language(A,B),
           const(B,vb)))).

Sentence:
which jobs are for bsee majors with at least 5 years
experience in windows nt ?
Logical Query:
answer(A, (job(A),req_deg(A,B),const(B,BSEE),req_exp(A,C),
           const(C,5),platform(A,D),const(D,windows nt)))).
```

```
Sentence:
what are the positions within hp that pay 40000 per year ?
Logical Query:
answer(A, (job(A),company(A,C),const(C,HP),
           salary_greater_than(A,40000,year)))).
```

```
Sentence:
what are the positions within dell that requires bscs ?
Logical Query:
answer(A, (job(A),company(A,C),const(C,Dell),
           req_deg(A,S),const(S,BSCS)))).
```

```
Sentence:
what jobs in boston have openings for a vb programmer ?
Logical Query:
answer(A, (job(A),loc(A,C),const(C,boston),language(A,L),
           const(L,vb)))).
```

Appendix D

Sample Theories Learned on Classifying Murder-for-hire Events

A couple of sample theories learned by each of the ILP systems: BETH, M-FOIL, and ALEPH are shown here. In each theory shown, each rule is preceeded by two numbers: 1) the number of positive examples covered by the rule, and 2) the number of negative examples covered by it. For example, the following rule:

```
[Rule 1] 32-1:
murder_for_hire(A) :-
  perpetrator(A,B),
  deliberateActors(C,B),
  murder(A),
  toPossessor(D,B),
```

```
  payer(D,E),
  recipientOfinfo(F,E),
  perpetrator(G,B),
  eventOccursAt(G,H),
  geographicalSubRegions(I,H),
  operatesinRegion(J,I).
```

covers thirty two positive examples and one negative example.

D.1 Sample Theories Learned By Beth

This is a theory learned by Beth:

```
[Rule 1] 32-1:
murder_for_hire(A) :-
  perpetrator(A,B),
  deliberateActors(C,B),
  murder(A),
  toPossessor(D,B),
  payer(D,E),
  recipientOfinfo(F,E),
  perpetrator(G,B),
  eventOccursAt(G,H),
  geographicalSubRegions(I,H),
  operatesinRegion(J,I).
```

```
[Rule 2] 19-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  ceo(C,B).

[Rule 3] 11-0:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  recipientOfinfo(D,C),
  hasMembers(E,C),
  orgHitman(E,F).

[Rule 4] 31-2:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  toPossessor(C,B),
  senderOfinfo(D,B),
  payer(C,E),
  deliberateActors(F,E),
  perpetrator(G,B),
```

```
  eventOccursAt(A,H),
  geographicalSubRegions(I,H),
  operatesinRegion(J,I).

[Rule 5] 11-0:
murder_for_hire(A) :-
  subEvents(B,A),
  subEvents(C,B).

[Rule 6] 30-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  objectsObserved(C,B),
  employees(D,B).

[Rule 7] 44-2:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  socialParticipants(C,B),
  toPossessor(D,B),
  payer(D,E),
  socialParticipants(F,E),
```

```
  payer(G,E),
  toPossessor(G,H),
  eventOccursAt(A,I),
  geographicalSubRegions(J,I).

[Rule 8] 24-0:
murder_for_hire(A) :-
  perpetrator(A,B),
  hitman(C,B),
  murder(A).

[Rule 9] 25-0:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  crimeVictim(B,D),
  employees(E,D).

[Rule 10] 4-0:
murder_for_hire(A) :-
murder(A),
  crimeVictim(A,B),
  senderOfinfo(C,B),
```

```
  hasMembers(D,B).

[Rule 11] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  orgHitman(C,B).

[Rule 12] 16-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  recipientOfinfo(D,B),
  socialParticipants(E,B),
  iteillocutionaryForce(D,F),
  recipientOfinfo(C,G),
  hasMembers(H,G),
  mafiyaGroup_Russian(H).

[Rule 13] 9-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
```

```
  orgHitman(C,B),
  deliberateActors(D,B).

[Rule 14] 10-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  toPossessor(C,B),
  senderOfinfo(D,B),
  hasMembers(E,B),
  orgHitman(E,F).

[Rule 15] 3-0:
murder_for_hire(A) :-
  perpetrator(A,B),
  orgHitman(C,B),
  murder(A),
  orgMiddleman(C,D),
  deliberateActors(E,D).

[Rule 16] 12-0:
murder_for_hire(A) :-
murder(A),
  perpetrator(A,B),
```

```
  senderOfinfo(C,B),
  eMailSending(C),
  iteillocutionaryForce(C,D),
  recipientOfinfo(C,E),
  crimeVictim(A,F),
  hasMembers(G,F).

[Rule 17] 6-0:
murder_for_hire(A) :-
  subEvents(B,A),
  perpetrator(B,C),
  hasMembers(D,C),
  hasMembers(D,E),
  hitman(F,E).

[Rule 18] 14-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  senderOfinfo(C,D),
  toPossessor(E,D),
  hasMembers(F,D).
```

```
[Rule 19] 10-0:
murder_for_hire(A) :-
  subEvents(B,A),
  perpetrator(B,C),
  hasMembers(D,C),
  mafiyaGroup_Russian(D),
  hasMembers(D,E),
  senderOfinfo(F,E),
  orgMiddleman(D,G).

[Rule 20] 9-3:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  senderOfinfo(C,D),
  toPossessor(E,D),
  socialParticipants(F,D),
  socialParticipants(F,G),
  payer(E,G),
  eventOccursAt(A,H),
  geographicalSubRegions(I,H).
```

This is another theory learned by BETH:

```
[Rule 1] 12-0:
```

```
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  deliberateActors(D,C),
  perpetrator(E,C),
  observing(E),
  recipientOfinfo(F,C).

[Rule 2] 32-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  objectsObserved(C,B),
  employees(D,B).

[Rule 3] 25-0:
murder_for_hire(A) :-
  perpetrator(A,B),
  hitman(C,B),
  murder(A).

[Rule 4] 8-0:
murder_for_hire(A) :-
```

```
  perpetrator(A,B),
  deliberateActors(C,B),
  murder(A),
  orgHitman(D,B).

[Rule 5] 11-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  iteillocutionaryForce(C,D),
  hasMembers(E,B),
  mafiyaGroup_Russian(E),
  orgHitman(E,F).

[Rule 6] 21-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  deliberateActors(C,B),
  senderOfinfo(D,B),
  recipientOfinfo(D,E),
  crimeVictim(A,F),
  hasMembers(G,F).
```

```
[Rule 7] 4-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  senderOfinfo(C,B),
  eMailSending(C),
  recipientOfinfo(C,D),
  crimeVictim(E,D).

[Rule 8] 4-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  senderOfinfo(C,B),
  hasMembers(D,B).

[Rule 9] 18-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  ceo(C,B).

[Rule 10] 44-2:
```

```
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  toPossessor(C,B),
  payer(C,D),
  recipientOfinfo(E,D),
  eMailSending(E),
  recipientOfinfo(F,D),
  senderOfinfo(F,G),
  perpetrator(H,B),
  eventOccursAt(A,I).

[Rule 11] 4-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  senderOfinfo(C,D),
  orgMiddleman(E,D).

[Rule 12] 5-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
```

```
  recipientOfinfo(C,B),
  subEvents(D,A),
  perpetrator(D,E),
  hasMembers(F,E).

[Rule 13] 4-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  deliberateActors(C,B),
  payer(D,B).

[Rule 14] 2-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  deliberateActors(C,B),
  subEvents(D,C).

[Rule 15] 22-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  deliberateActors(C,B),
```

```
  perpetrator(D,B),
  objectsObserved(D,E),
  hasMembers(F,E),
  crimeVictim(A,E).

[Rule 16] 5-0:
murder_for_hire(A) :-
  perpetrator(A,B),
  orgHitman(C,B),
  murder(A),
  orgMiddleman(C,D),
  orgHitman(C,E),
  agentPhoneNumber(B,F),
  receiverNumber(G,F).

[Rule 17] 26-1:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  socialParticipants(C,B),
  toPossessor(D,B),
  perpetrator(E,B),
  objectsObserved(E,F),
  crimeVictim(A,F),
```

```
  socialParticipants(C,G),
  recipientOfinfo(H,G),
  senderOfinfo(H,I).

[Rule 18] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  orgHitman(C,B).

[Rule 19] 6-1:
murder_for_hire(A) :-
  perpetrator(A,B),
  orgHitman(C,B),
  murder(A),
  senderOfinfo(D,B),
  recipientOfinfo(E,B),
  senderOfinfo(E,F),
  recipientOfinfo(G,F),
  iteillocutionaryForce(E,H),
  iteillocutionaryForce(I,H),
  dateOfEvent(G,J).

[Rule 20] 15-0:
```

```
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  senderOfinfo(C,D),
  toPossessor(E,D),
  hasMembers(F,D).

[Rule 21] 20-5:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  senderOfinfo(C,D),
  socialParticipants(E,D),
  eMailSending(C),
  socialParticipants(E,F),
  recipientOfinfo(G,D),
  iteillocutionaryForce(G,H),
  eventOccursAt(A,I).

[Rule 22] 1-0:
murder_for_hire(A) :-
  murder(A),
```

```
  crimeVictim(A,B),
  orgMiddleman(C,B).

[Rule 23] 7-0:
murder_for_hire(A) :-
  subEvents(B,A),
  premeditatedMurder(B),
  perpetrator(B,C),
  agentPhoneNumber(C,D),
  receiverNumber(E,D),
  hasMembers(F,C),
  mafiyaGroup_Russian(F),
  orgMiddleman(F,G).
```

D.2 Sample Theories Learned By m-Foil

This is a theory learned by m-Foil:

```
[Rule 1] 11-0:
murder_for_hire(A) :-
  subEvents(B,A),
  subEvents(C,B),
  perpetrator(B,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,E),
  geographicalSubRegions(F,G),
```

```
  eventOccursAt(A,G),
  deviceTypeused(A,H),
  deviceTypeused(A,I),
  murder(A).

[Rule 2] 24-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  hitman(C,B).

[Rule 3] 13-0:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  deliberateActors(D,C),
  perpetrator(B,E),
  deliberateActors(D,F),
  hasMembers(G,F),
  hasMembers(G,H),
  mafiyaGroup_Russian(G),
  eventOccursAt(A,I).
```

```
[Rule 4] 19-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  ceo(C,B),
  hasMembers(D,B),
  employees(C,E),
  crimeVictim(A,F),
  ceo(C,F),
  eventOccursAt(A,G),
  eventOccursAt(A,H),
  geographicalSubRegions(I,H).

[Rule 5] 9-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  deliberateActors(C,B),
  orgHitman(D,B),
  mafiyaGroup_Russian(D),
  orgHitman(D,E),
  hasMembers(D,F),
  eventOccursAt(A,G),
  eventOccursAt(A,H),
```

```
  geographicalSubRegions(I,G).

[Rule 6] 43-7:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  deliberateActors(C,B),
  perpetrator(A,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,E),
  geographicalSubRegions(G,E),
  eventOccursAt(A,H),
  deviceTypeused(A,I),
  deviceTypeused(A,J).

[Rule 7] 3-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  orgHitman(C,B),
  orgMiddleman(C,D),
  deliberateActors(E,D),
  hasMembers(C,B),
  hasMembers(C,F),
```

```
  mafiyaGroup_Russian(C),
  orgHitman(C,G),
  eventOccursAt(A,H).

[Rule 8] 4-2:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  deliberateActors(C,B),
  eventOccursAt(A,D),
  geographicalSubRegions(E,D),
  geographicalSubRegions(F,D),
  eventOccursAt(A,G),
  operatesinRegion(H,E),
  hasMembers(H,I),
  crimeVictim(J,B).

[Rule 9] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  orgHitman(C,B),
  person(B),
  toPossessor(D,B),
```

```
  paying(D),
  perpetrator(A,E),
  orgHitman(C,F),
  deviceTypeused(A,G),
  deviceTypeused(H,G).

[Rule 10] 30-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  objectsObserved(C,B),
  employees(D,B),
  hasMembers(E,B),
  employees(E,B),
  hasMembers(F,D),
  objectsObserved(G,B),
  observing(G),
  observing(C).

[Rule 11] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  perpetrator(C,B),
```

```
  murder(C),
  perpetrator(C,D),
  crimeVictim(A,D),
  toPossessor(E,B),
  senderOfinfo(F,D),
  person(D),
  person(B).

[Rule 12] 6-1:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  orgHitman(D,C),
  orgHitman(D,E),
  hasMembers(D,E),
  hasMembers(D,F),
  eventOccursAt(A,G),
  eventOccursAt(A,H),
  geographicalSubRegions(I,G).

[Rule 13] 4-0:
murder_for_hire(A) :-
  murder(A),
```

```
  perpetrator(A,B),
  recipientOfinfo(C,B),
  hasMembers(D,B),
  hasMembers(D,E),
  deliberateActors(F,E),
  orgMiddleman(D,E),
  orgMiddleman(G,E),
  orgHitman(G,H),
  mafiyaGroup_Russian(G).

[Rule 14] 1-2:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  crimeVictim(C,B),
  eventOccursAt(A,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,D),
  geographicalSubRegions(G,D),
  operatesinRegion(H,F),
  hasMembers(I,H),
  hasMembers(J,H).

[Rule 15] 10-2:
```

```
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  socialParticipants(C,B),
  recipientOfinfo(D,B),
  eMailSending(D),
  iteillocutionaryForce(D,E),
  person(B),
  recipientOfinfo(F,B),
  perpetrator(G,B),
  iteillocutionaryForce(H,E).

[Rule 16] 23-2:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  hasMembers(D,B),
  hasMembers(D,E),
  mafiyaGroup_Russian(D),
  iteillocutionaryForce(C,F),
  iteillocutionaryForce(C,G),
  eventOccursAt(A,H),
  eventOccursAt(A,I).
```

```
[Rule 17] 4-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  eMailSending(C),
  hasMembers(D,B),
  orgHitman(D,E),
  hasMembers(D,E),
  perpetrator(A,F),
  person(F),
  person(E).

[Rule 18] 1-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  iteillocutionaryForce(C,D),
  eMailSending(C),
  recipientOfinfo(C,E),
  employees(F,E),
  hasMembers(G,E),
```

```
  employees(G,E),
  hasMembers(H,F).

[Rule 19] 24-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  recipientOfinfo(C,D),
  deliberateActors(E,D),
  hasMembers(F,D),
  hasMembers(F,G),
  deliberateActors(E,G),
  deviceTypeused(A,H),
  deviceTypeused(A,I).

[Rule 20] 8-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  toPossessor(C,B),
  recipientOfinfo(D,B),
  eMailSending(D),
  perpetrator(E,B),
```

```
  objectsObserved(E,F),
  crimeVictim(A,F),
  objectsObserved(G,F),
  observing(G).

[Rule 21] 14-6:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  eMailSending(C),
  iteillocutionaryForce(C,D),
  recipientOfinfo(C,E),
  perpetrator(A,F),
  person(E),
  person(B),
  recipientOfinfo(C,G).

[Rule 22] 2-0:
murder_for_hire(A) :-
  subEvents(B,A),
  perpetrator(B,C),
  hasMembers(D,C),
  hasMembers(D,E),
```

```
  mafiyaGroup_Russian(D),
  orgMiddleman(D,F),
  operatesinRegion(D,G),
  geographicalSubRegions(G,H),
  eventOccursAt(A,H),
  geographicalSubRegions(I,H).
```

```
[Rule 23] 21-6:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  eMailSending(C),
  senderOfinfo(C,D),
  person(B),
  recipientOfinfo(E,D),
  iteillocutionaryForce(E,F),
  perpetrator(A,G),
  person(G).
```

This is another theory learned by M-FOIL:

```
[Rule 1] 13-0:
murder_for_hire(A) :-
  subEvents(B,A),
  subEvents(C,B),
```

```
  perpetrator(B,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,E),
  geographicalSubRegions(F,G),
  eventOccursAt(A,G),
  deviceTypeused(A,H),
  deviceTypeused(A,I),
  murder(A).

[Rule 2] 25-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  hitman(C,B).

[Rule 3] 18-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  ceo(C,B).

[Rule 4] 46-6:
murder_for_hire(A) :-
  murder(A),
```

```
  perpetrator(A,B),
  deliberateActors(C,B),
  perpetrator(A,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,E),
  geographicalSubRegions(G,E),
  eventOccursAt(A,H),
  deviceTypeused(A,I),
  deviceTypeused(A,J).

[Rule 5] 2-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  deliberateActors(C,B),
  person(B),
  planningTodoSomething(C),
  recipientOfinfo(D,B),
  eMailSending(D),
  iteillocutionaryForce(D,E),
  iteillocutionaryForce(D,F),
  perpetrator(A,G).

[Rule 6] 3-1:
```

```
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  orgHitman(C,B),
  orgMiddleman(C,D),
  orgHitman(C,E),
  hasMembers(C,F),
  hasMembers(G,E),
  mafiyaGroup_Russian(C),
  orgMiddleman(G,H),
  eventOccursAt(A,I).

[Rule 7] 1-2:
murder_for_hire(A) :-
  perpetrator(A,B),
  crimeVictim(C,B),
  eventOccursAt(A,D),
  eventOccursAt(A,E),
  geographicalSubRegions(F,D),
  geographicalSubRegions(G,D),
  operatesinRegion(H,F),
  hasMembers(I,H),
  hasMembers(J,H),
  industry_Localized(J).
```

```
[Rule 8] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  orgMiddleman(C,B),
  perpetrator(A,D),
  orgMiddleman(E,B),
  mafiyaGroup_Russian(C),
  objectsObserved(F,B),
  objectsObserved(G,B),
  observing(G),
  observing(F).

[Rule 9] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  orgHitman(C,B),
  person(B),
  toPossessor(D,B),
  paying(D),
  perpetrator(A,E),
  orgHitman(C,F),
```

```
  deviceTypeused(A,G),
  deviceTypeused(H,G).

[Rule 10] 5-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  hasMembers(D,B),
  iteillocutionaryForce(C,E),
  hasMembers(D,F),
  crimeVictim(G,F),
  deviceTypeused(A,H),
  deviceTypeused(A,I),
  eventOccursAt(A,J).

[Rule 11] 32-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  objectsObserved(C,B),
  employees(D,B),
  hasMembers(E,B),
  employees(E,B),
```

```
  hasMembers(F,D),
  objectsObserved(G,B),
  observing(G),
  observing(C).

[Rule 12] 1-0:
murder_for_hire(A) :-
  murder(A),
  crimeVictim(A,B),
  perpetrator(C,B),
  murder(C),
  perpetrator(C,D),
  crimeVictim(A,D),
  toPossessor(E,B),
  senderOfinfo(F,D),
  person(D),
  person(B).

[Rule 13] 1-0:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
  perpetrator(B,C),
  hasMembers(D,C),
```

```
  hasMembers(D,E),
  objectsObserved(F,E),
  mafiyaGroup_Russian(D),
  objectsObserved(F,G),
  deliberateActors(H,G),
  deliberateActors(I,G).

[Rule 14] 7-1:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  orgHitman(D,B),
  perpetrator(A,E),
  orgHitman(F,E),
  mafiyaGroup_Russian(D),
  person(E),
  person(B),
  mafiyaGroup_Russian(F).

[Rule 15] 12-7:
murder_for_hire(A) :-
  murder(A),
  subEvents(B,A),
```

```
  subEvents(B,C),
  murder(C),
  premeditatedMurder(B),
  perpetrator(B,D),
  perpetrator(A,D),
  recipientOfinfo(E,D),
  eMailSending(E),
  eventOccursAt(A,F).

[Rule 16] 2-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  senderOfinfo(C,B),
  hasMembers(D,B),
  eMailSending(C),
  orgHitman(D,E),
  hasMembers(D,E),
  perpetrator(A,F),
  person(F),
  person(E).

[Rule 17] 41-14:
murder_for_hire(A) :-
```

```
  murder(A),
  perpetrator(A,B),
  socialParticipants(C,B),
  person(B),
  senderOfinfo(D,B),
  perpetrator(A,E),
  iteillocutionaryForce(D,F),
  iteillocutionaryForce(D,G),
  meetingTakingPlace(C),
  person(E).

[Rule 18] 16-0:
murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  toPossessor(C,B),
  recipientOfinfo(D,B),
  perpetrator(E,B),
  objectsObserved(E,F),
  hasMembers(G,F),
  crimeVictim(A,F),
  hasMembers(G,H),
  dateOfEvent(D,I).
```

```
[Rule 19] 2-0:

murder_for_hire(A) :-
  subEvents(B,A),
  perpetrator(B,C),
  hasMembers(D,C),
  hasMembers(D,E),
  mafiyaGroup_Russian(D),
  orgMiddleman(D,F),
  operatesinRegion(D,G),
  geographicalSubRegions(G,H),
  eventOccursAt(A,H),
  geographicalSubRegions(I,H).

[Rule 20] 20-7:

murder_for_hire(A) :-
  murder(A),
  perpetrator(A,B),
  recipientOfinfo(C,B),
  eMailSending(C),
  senderOfinfo(C,D),
  person(B),
  recipientOfinfo(E,D),
  iteillocutionaryForce(E,F),
  perpetrator(A,G),
```

```
  person(G).
```

D.3 Sample Theories Learned By ALEPH

This is a theory learned by ALEPH:

```
[Rule 1] 37-1:
ckmurder(A) :-
  eventOccursAt(A,B),
  perpetrator(A,C),
  deliberateActors(D,C),
  toPossessor(E,C),
  payer(E,F),
  agentPhoneNumber(F,G),
  receiverNumber(H,G).

[Rule 2] 87-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  crimeVictim(A,C),
  hasMembers(D,C).

[Rule 3] 14-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  perpetrator(A,C),
```

```
  orgHitman(D,C),
  agentPhoneNumber(C,E),
  receiverNumber(F,E).

[Rule 4] 9-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  crimeVictim(A,C),
  objectsObserved(D,C),
  perpetrator(A,E),
  socialParticipants(F,E),
  socialParticipants(F,G),
  hasMembers(H,G),
  orgMiddleman(H,I).

[Rule 5] 30-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  perpetrator(A,C),
  toPossessor(D,C),
  senderOfinfo(E,C),
  recipientOfinfo(E,F),
  deliberateActors(G,F),
  dateOfEvent(E,H),
```

```
  dateOfEvent(I,H),
  eMailSending(I).


[Rule 6] 4-0-
ckmurder(A) :-
  eventOccursAt(A,B),
  crimeVictim(A,C),
  toPossessor(D,C),
  payer(D,E),
  hasMembers(F,E),
  perpetrator(A,G),
  toPossessor(H,G).
```

This is another theory learned by ALEPH:

```
[Rule 1] 71-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  perpetrator(D,E),
  socialParticipants(F,E),
  crimeVictim(A,G),
  hasMembers(H,G).


[Rule 2] 2-0:
```

```
ckmurder(A) :-
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  subEvents(E,D),
  crimeVictim(D,F),
  senderOfinfo(G,F),
  perpetrator(A,H),
  toPossessor(I,H).

[Rule 3] 3-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  subEvents(E,D),
  crimeVictim(D,F),
  hasMembers(G,F),
  orgMiddleman(G,H),
  perpetrator(A,I),
  toPossessor(J,I).

[Rule 4] 12-0:
ckmurder(A) :-
```

```
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  perpetrator(D,E),
  employees(F,E),
  perpetrator(A,G),
  hasMembers(H,G),
  orgHitman(H,I),
  socialParticipants(J,I).

[Rule 5] 8-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  crimeVictim(A,E),
  senderOfinfo(F,E),
  eMailSending(F),
  perpetrator(A,G),
  agentPhoneNumber(G,H),
  receiverNumber(I,H).

[Rule 6] 14-0:
ckmurder(A) :-
```

```
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  subEvents(D,E),
  observing(E),
  crimeVictim(D,F),
  socialParticipants(G,F),
  perpetrator(A,H),
  toPossessor(I,H).

[Rule 7] 5-0:
ckmurder(A) :-
  eventOccursAt(A,B),
  eventOccursAt(C,B),
  subEvents(D,C),
  subEvents(D,E),
  observing(E),
  crimeVictim(A,F),
  recipientOfinfo(G,F),
  senderOfinfo(G,H),
  crimeVictim(I,H).

[Rule 8] 78-0:
ckmurder(A) :-
```

```
eventOccursAt(A,B),
eventOccursAt(C,B),
subEvents(D,C),
subEvents(D,E),
observing(E),
crimeVictim(A,F),
hasMembers(G,F).
```

Appendix E

The Learning Curve for CHILL

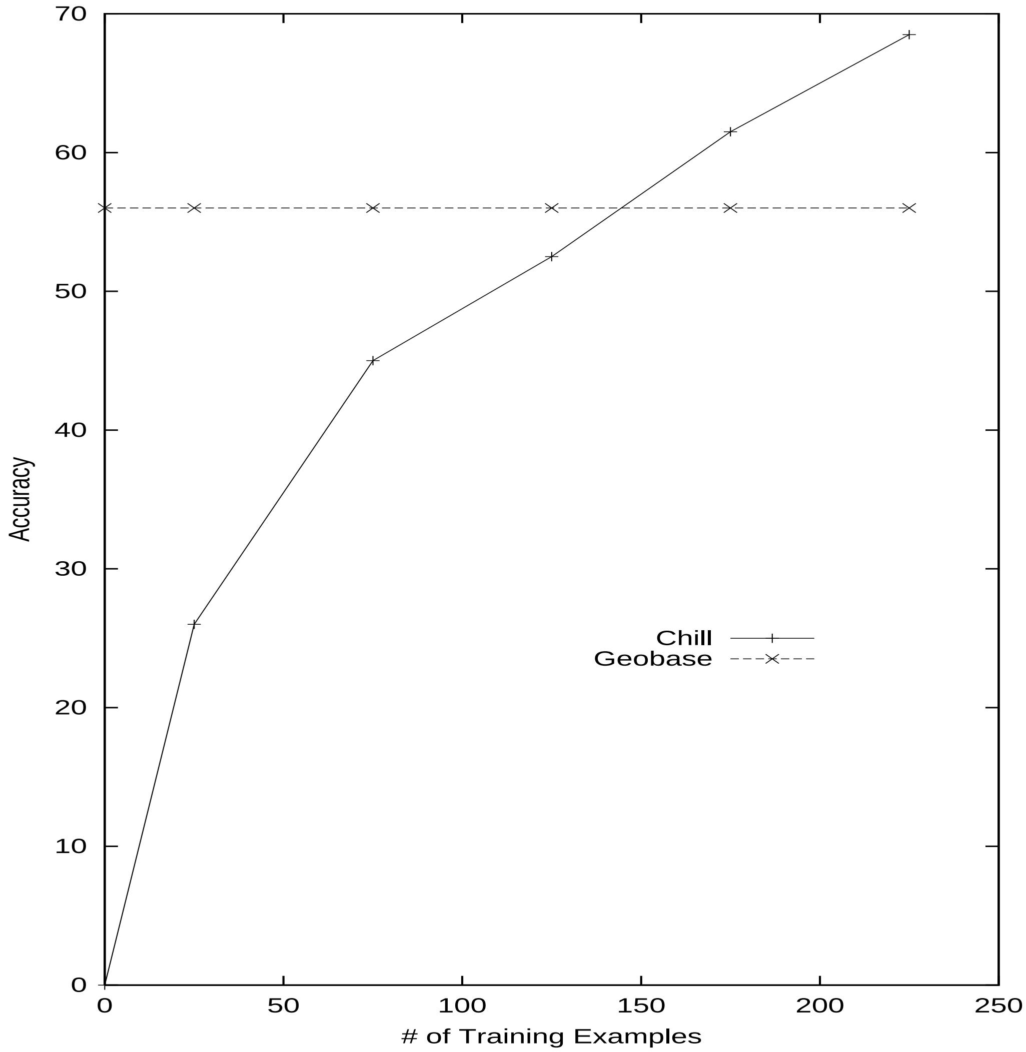


Figure E.1: Geoquery: Accuracy

Appendix F

Learning curves and Training Time on Link Discovery

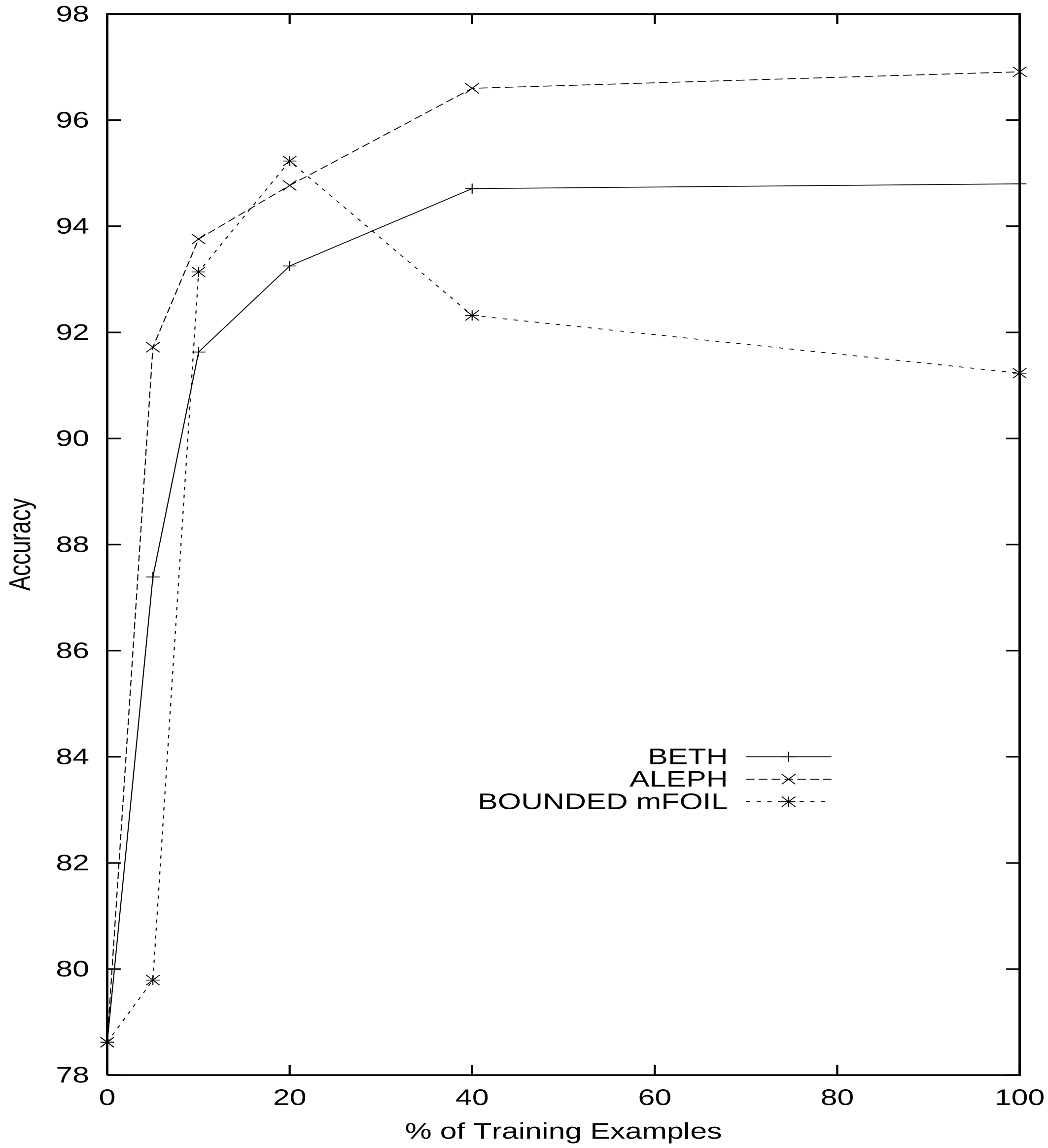


Figure F.1: Learning curves

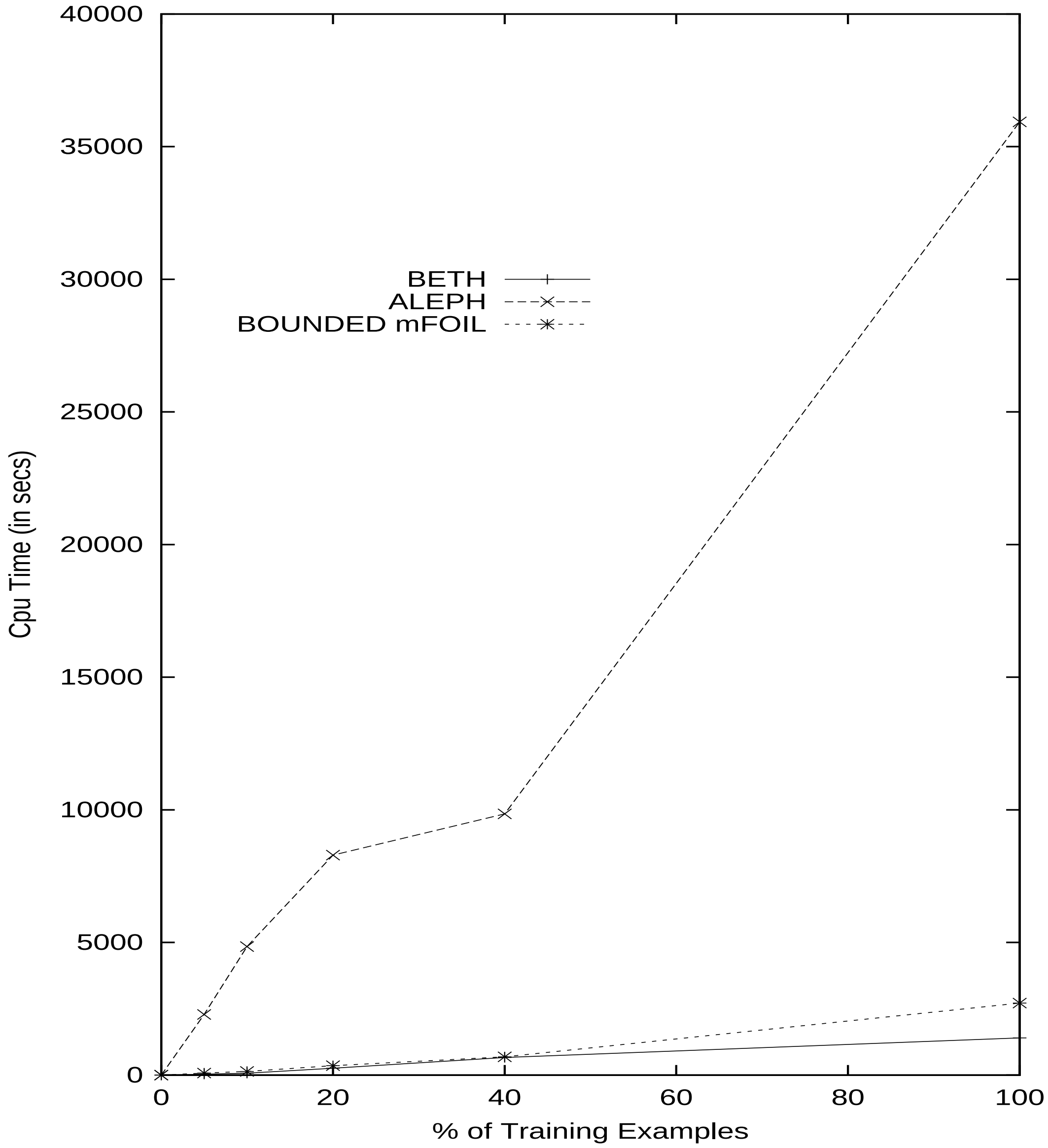


Figure F.2: Training time

Bibliography

Abramson, H., & Dahl, V. (1989). *Logic Grammars.* Springer Verlag, New York.

Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 207–216 New York.

Allen, J. F. (1995). *Natural Language Understanding (2nd Ed.).* Benjamin/Cummings, Menlo Park, CA.

Armstrong-Warwick, S. (1993). Preface (to the special issue on using large corpora). *Computational Linguistics*, *19*(1), iii–iv.

Bahl, L. R., Jelinek, F., & Mercer, R. L. (1983). A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *5*(2), 179–190.

Bain, M., & Muggleton, S. (1992). Non-monotonic learning. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 145–162. Academic Press, New York, NY.

Berwick, R. C. (1985). *The Acquisition of Syntactic Knowledge.* MIT Press, Cambridge, MA.

Blockeel, H., Demoen, B., Janssens, G., Vandecasteele, H., & Laer, W. V. (2000). Two advanced transformations for improving the efficiency of an ILP system (work-in-progress reports). In *10th International Conference on Inductive Logic Programming*, pp. 43–59 London, UK.

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Reading:MA.

Brown, J. S., & Burton, R. R. (1975). Multiple representations of knowledge for tutorial reasoning. In Bobrow, D. G., & Collins, A. (Eds.), *Representation and Understanding.* Academic Press, New York.

Califf, M. E., & Mooney, R. J. (1999). Relational learning of pattern-match rules for information extraction. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pp. 328–334 Orlando, FL.

Cameron-Jones, R. M., & Quinlan, J. R. (1994). Efficient top-down induction of logic programs. *SIGART Bulletin*, *5*(1), 33–42.

Cestnik, B. (1990). Estimating probabilities: A crucial task in machine learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pp. 147–149 Stockholm, Sweden.

Charniak, E. (1996). Tree-bank grammars. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 1031–1036 Portland, OR.

Charniak, E., Hendrickson, C., Jacobson, N., & Perkowitz, M. (1993). Equations for part-of-speech tagging. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 784–789 Washington, D.C.

Collins, M. J. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL-97)*, pp. 16–23.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pp. 151–158 New York. Association for Computing Machinery.

Costa, V. S., Srinivasan, A., Camacho, R. C., Blockeel, H., B.Demoen, Janssens, G., Struyf, J., Vandecasteele, H., & Laer, W. V. (2002). Query transformations for improving the efficiency of ILP systems (to appear). *Journal of Machine Learning Research*.

Doets, K. (Ed.). (1994). *From Logic to Logic Programming*. The MIT Press, Cambridge, Massachusetts.

Džeroski, S., & Lavrač, N. (2001). An introduction to inductive logic programming. In Džeroski, S., & Lavrač, N. (Eds.), *Relational Data Mining*. Springer Verlag, Berlin.

Finn, P. W., Muggleton, S., Page, D., & Srinivasan, A. (1998). Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, *30*(2-3), 241–270.

Hendrix, G. G., Sacerdoti, E., Sagalowicz, D., & Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, *3*(2), 105–147.

Hirschberg, J. (1998). Every time I fire a linguist, my performance goes up, and other myths of the statistical natural language processing revolution. Invited talk, Fifteenth National Conference on Artificial Intelligence (AAAI-98).

Kersting, K., & Raedt, L. D. (2001). Adaptive bayesian logic programs. In *11th International Conference on Inductive Logic Programming*, pp. 104–117 Strasbourg, France.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, *1*, 4–7.

Kuhn, R., & De Mori, R. (1995). The application of semantic classification trees to natural language understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *17*(5), 449–460.

Lavrac, N., & Dzeroski, S. (Eds.). (1994a). *Inductive Logic Programming: Techniques and Application.* Ellis Horwood, Chichester.

Lavrac, N., & Dzeroski, S. (1994b). *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

Litman, D. J. (1996). Cue phrase classification using machine learning. *Journal of Artificial Intelligence Research*, *5*, 53–95.

Lloyd, J. W. (1984). Foundations of logic programming. *Symbolic Computation.*

Manning, C. D., & Carpenter, B. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the Fifth International Workshop on Parsing Technologies*, pp. 147–158.

Melville, P., & Mooney, R. J. (2003). Constructing diverse classifier ensembles using artificial training examples. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2003) (To appear)* Acapulco, Mexico.

Michalski, R. S. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, *20*, 111–161.

Miller, S., Stallard, D., Bobrow, R., & Schwartz, R. (1996). A fully statistical approach to natural language interfaces. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL-96)*, pp. 55–61 Santa Cruz, CA.

Mitchell, T. (1997). *Machine Learning.* McGraw-Hill, New York, NY.

Mooney, R. J., Melville, P., Tang, L. R., Shavlik, J., de Castro Dutra, I., Page, D., & Costa, V. S. (2002). Relational data mining with inductive logic programming for link discovery. In *Proceedings of the National Science Foundation Workshop on Next Generation Data Mining.*

Muggleton, S. (1992). Inverting implication. In Muggleton, S. (Ed.), *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, pp. 19–39.

Muggleton, S. (1999). Inductive Logic Programming. In *The MIT Encyclopedia of the Cognitive Sciences (MITECS).* MIT Press.

Muggleton, S., & Buntine, W. (1988). Machine invention of first order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning (ICML-88)*, pp. 339–351. Morgan Kaufmann.

Muggleton, S., Srinivasan, A., & Bain, M. (1992). Compression, significance and accuracy. In Sleeman, D., & Edwards, P. (Eds.), *Proceedings of the 9th International Workshop on Machine Learning*, pp. 338–347. Morgan Kaufmann.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing Journal*, *13*, 245–286.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning (ICML-88)*, pp. 339–352 Ann Arbor, MI.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory* Tokyo, Japan. Ohmsha.

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 281–297. Academic Press, New York.

Muggleton, S. H., & Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19*, 629–679.

Pazzani, M. J., & Kibler, D. F. (1992). The utility of background knowledge in inductive learning. *Machine Learning*, *9*, 57–94.

Pereira, F. C. N., & Shabes, Y. (1992). Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL-92)*, pp. 128–135 Newark, Delaware.

Plotkin, G. D. (1970). A note on inductive generalisation. *Machine Intelligence*, *5*, 153–163.

Plotkin, G. D. (1971). A further note on inductive generalization. *Machine Intelligence*, *6*(101-124).

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo,CA.

Quinlan, J. R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Ratnaparkhi, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, *34*, 151–176.

Reeker, L. H. (1976). The computational study of language acquisition. In Yovits, M., & Rubinoff, M. (Eds.), *Advances in Computers*, Vol. 15, pp. 181–237. Academic Press, New York.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*, 465–471.

Robinson, J. A. (1983). A machine-oriented logic based on the resolution principle. In Siekmann, J., & Wrightson, G. (Eds.), *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pp. 397–415. Springer, Berlin, Heidelberg.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., & McClelland, J. L. (Eds.), *Parallel Distributed Processing, Vol. I*, pp. 318–362. MIT Press, Cambridge, MA.

Santos Costa, V. (1999). Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pp. 261–267. Springer-Verlag.

Siklossy, L. (1972). Natural language learning by computer. In Simon, H. A., & Siklossy, L. (Eds.), *Representation and meaning: Experiments with Information Processsing Systems*. Prentice Hall, Englewood Cliffs, NJ.

Tang, L. R., & Mooney, R. J. (2001). Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th European Conference on Machine Learning*, pp. 466–477 Freiburg, Germany.

Thompson, C. A., & Mooney, R. J. (1999). Automatic construction of semantic lexicons for learning natural language interfaces. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pp. 487–493 Orlando, FL.

Tomita, M. (1986). *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston.

Waltz, D. L. (1978). An English language question answering system for a large relational database. *Communications of the Association for Computing Machinery*, *21*(7), 526–539.

Warren, D. H. D., & Pereira, F. C. N. (1982). An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, *8*(3-4), 110–122.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the Association for Computing Machinery*, *13*, 591–606.

Zelezny, F., Srinivasan, A., & Page, D. (2002). Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the 12th International Conference on Inductive Logic Programming.*

Zelle, J. M. (1995). *Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers.* Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, TX. Also appears as Artificial Intelligence Laboratory Technical Report AI 96-249.

Zelle, J. M., & Mooney, R. J. (1993). Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 817–822 Washington, D.C.

Zelle, J. M., & Mooney, R. J. (1994). Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML-94)*, pp. 343–351 New Brunswick, NJ.

Vita

Lap Poon Rupert Tang was born in Hong Kong on March 29, 1973, the son of Koon Sing David and Siu Chun Helen Tang. After completing his work at Queen's College in Hong Kong, he entered the University of Texas at Austin, from which he received his Bachelor of Science in Computer Sciences in May, 1995, graduating cum laude. In the Fall semester of 1995, he entered the Ph.D. program of the Department of Computer Sciences at the University of Texas at Austin where he received his Master of Science in Computer Sciences in the summer of 1997. He will be working at the University of Delaware as a post doctoral fellow on September 1, 2003.

Permanent Address: 6 Tai Hang Drive
10A Park Garden
Hong Kong
China

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.